# Middleware Support for Voting and Value Checking (WORKING TITLE)

Zhiyuan Zhan

Microsoft Corporation

One Microsoft Way, Redmond, WA 98052

zhiyuan.zhan@microsoft.com

David E. Bakken

School of EECS

Washington State University, Pullman, WA 99163

bakken@eecs.wsu.edu

July 3, 2007

**Keywords:** Middleware, Voting, Data Fusion

**Abstract**

Let's put down something funny here as a temporary abstract

# 1 Introduction

**DAVEDO: Dave will provide this leadup soon...**

The research contributions of this paper are:

- An analysis of why byte-by-byte value comparison mechanisms used in voting middleware and elsewhere do not work in the presence of the kinds of heterogeneity that are inherent in distributed computing systems, the very heterogeneity that middleware is designed to mask programmers from.
- An analyis of the architectural implications of these limitations on active replication, byzantine quorum, data fusion, peer-to-peer, and other kinds of distributed architectures that must detect value falures and otherwise compare values from different sources.
- The design and implementation of the Voting Virtual Machine (VVM), a middleware component that votes correctly on application-level data and can be embedded into different middleware substrates or application programs.
- The design and implementation of the Voting Definition Language (VDL), which is used with the VVM and allows voting algorithms to be coded in a portable and reusable manner.
- An experimental evaluation of the performance of the VVM.
- An analysis of the positive and negative lessons learned about how voting and other data-comparison mechanisms can be provided in a virtual machine with voting-specific primitives.

The remainder of this paper is organized as follows. Section 2 analyzes why byte-byte-voting mechanisms do not work in the face of heterogeneity and its architectural implications. Section 3 describes the VVM design. Section 4 describes VDL, and Section 5 gives examples of common voting algorithms expressed in VDL. Section 6 describes the implementation of VVM and other supporting components and tools. Section 7 provides an evaluation of the performance of the VVM. Section 8 describes the lessons learned about providing voting and other data comparison mechanisms in a virtual machine providing specialized primitives. Section 9 describes related work, and Section 10 concludes.

# 2 Limitations of Current Voting Schemes

## 2.1 Data Sharing in Distributed Systems

Many distributed systems, such as distributed file systems, world wide web (WWW), distributed objects and P2P systems, allow distributed users to access shared data that is either cached or replicated at multiple nodes.

### 2.1.1 Distributed File Systems:

Many operating systems use caching to improve file system performances. Some of the popular distributed file systems are xFS [1], Spring [2], NFS [3], Coda [4], Farsite [5] [6] [7] and Ivy [8].

$xFS$ is a serverless file system, which provides strong ordering by ensuring that a single writer or multiple readers are able to access a file at a given time. It allows client nodes to cooperatively cache file blocks that are accessed. Any node in the system can cache data and supply it to other clients. The location independence of such a configuration with fast local-area networks gives better performance and scalability than the traditional systems.

The $Spring$ file system supports cache coherent file data and attributes. It uses the virtual memory system to cache data and keep them consistent. It provides two types of file servers to accomplish the task: one that provide coherent access to files they export, and the other that runs on each machine to provide read and write operations for cached data.

The $NFS$ system allows multiple clients to access files located at one server. The server is stateless and does not maintain any information on either the clients or the way the file is cached. Therefore, all modifications of the file must be written back to the server once the file cache is closed at the client side. Under "sequential write sharing" (meaning a file cannot be open simultaneously for both reading and writing at the same time in different clients), each client reads the most recent copy of the file. However, NFS does not provide mechanisms to ensure such "sequential write sharing", which may result in client reading stale data.

$Farsite$ is a scalable, decentralized file system. It uses a set of insecure and loosely coupled machines to implement a P2P file system which is secure and reliable. Updates in Farsite are maintained by lazy propagation and content leases. Strong consistency of the file

system is maintained by means of leases. For example, a write/read lease has to be obtained before a client can modify/observe the content of a file. Multiple read leases of the same file is allowed, but only one write lease can be granted with no other leases (whether read or write) granted for the same file.

$Ivy$ is distributed P2P file system which supports both read and writes. It provides NFS-like interfaces to application users. Ivy solely consists of a set of logs, stored in distributed hash table (DHash). When network partition happens, the application specific conflict resolvers are used to resolve the update conflicts.

$Coda$ is a distributed file system that provides server and network failure tolerance. It uses server replication and disconnected operations to achieve those goals. The replication unit in Coda is called a *volume*. A client that accesses the *accessible volume storage group* (AVSG) takes care of the update dissemination and implements the "read one, write all" logic, in order to minimize the burden at server side. Network partition is to some extent tolerated by allowing continued operations on client side cached copy and update resolution is applied when client goes back online and submits its offline updates.

Note that most distributed file systems introduced above need to provide a guarantee that the file is accessed in a way that is the same as if the file is stored in a centralized server. This implies strong consistency requirement. In Ivy and Coda, updates are allowed even when a system partition happens. These systems depend on conflicts resolution when different partitions are connected back together. Our information sharing system is different from the distributed file system in that it generally can tolerate relaxed consistency requirements based on different user needs. Therefore, to provide a "single copy as if being stored in a centralized server" is not part of our goal. Similarly, to provide support for different consistency levels of the same file in the system at the same time is not the goal of distributed file systems.

### 2.1.2 World Wide Web:

Consistency protocols for web caching are described in [9], [10], [11]. Weak consistency protocols based on time to live (TTL) mechanism are presented in [10] and [9]. However, those weak consistency models focus more on providing better scalability and enhancing system performance rather than preventing clients from reading stale data. A stronger notion of

consistency based on invalidation and pull is presented in [11]. Generally the consistency requirements in WWW literature mainly focus on reducing the staleness of the cached copy at the client side. In our system, however, reading stale data is allowed, as long as the updates follow certain order that is guaranteed by the system. We focus more on the ordering aspect of the consistency requirement other than timeliness.

### 2.1.3 Distributed Objects:

$CORBA$ [12], [13], [14], [15] stands for Common Object Request Broker Architecture, which is Object Management Group (OMG [12])'s open client-server middleware architecture used for applications to access services across networks. CORBA's replication service [16], [17] supports object replication in order to improve performance and provide a mechanism for fault tolerance. The replicas of the same object are kept in strongly consistent state as the requests are totally and causally ordered across all replicas [17].

Object caching has also been studied in systems such as Thor [18], Rover [19], Bayou [20] and others. The $Thor$ system assumes a transaction notion for the operations on cached objects and validates them before they are committed. The $Rover$ system was developed to address the problem of bandwidth constraints and possible disconnection of clients. The client's copy will be serialized by the system when the client goes back online (or has enough bandwidth). The users have to manually resolve any possible conflicts. $Bayou$ is a replicated, weakly consistent storage system designed for a mobile computing environment where disconnection is possible. Bayou has focused on supporting application-specific mechanisms to detect and resolve the update conflicts. It proposed two techniques for such support: *dependency checks* and *merge procedures*. In addition, the updates are propagated in Bayou by pair-wise anti-entropy in order to guarantee eventual consistency. To our knowledge, the above distributed objects system all provide the same level of consistency to all replicas in the system. None of them allow for replicas of the same object to be running at different consistency level at the same time.

### 2.1.4 Peer-to-peer Information Sharing:

Peer-to-peer (P2P) data sharing systems have become popular in recent years [21] [22] [23]. Generally data items shared by P2P systems are considered to be fairly static and updates seldom happen. For example, in P2P email system described in [24], email components are stored in distributed nodes, and they support only read and delete operations. No updates by the write operation will be generated (so strong consistency requirement is not needed). If updates do happen, the modified data item is usually considered as a newer version of the old one and both versions co-exist in the system. Typically, the directory service maintains an entry of each version for centralized P2P systems such as Gnutella [23] [25] and new updates do not get propagated among replicated data items in such systems. Updates do get propagated in some P2P systems designed to be "writable", like FreeNet [26] and OceanStore [27]. However, they view P2P systems as a homogeneous system and do not consider the resource and application behavior heterogeneity. Furthermore, consistency guarantees in those systems are limited. When used as an update dissemination mechanism for in P2P systems, our work on mixed consistency model and agile dissemination is more flexible and can provide different consistency guarantees to different peers. Finally, our system supports multiple writers of the same object, while most of the P2P systems (e.g. FreeNet [26] and PAST [28]) do not.

Figure 1 is copied from Dave's slides.

# 3 Voting Virtual Machine Architecture

## 3.1 Overview

We have designed and developed a prototype version of the Voting Virtual Machine (VVM), and its companion Voting Definition Language (VDL) policy language [29–31].

The VVM provides a general, adaptable and portable mechanism for voting. It can be used standalone to develop and test portable voting algorithms. It supports adaptive voting, which allows different voting policies (pieces of compiled VDL code) be loaded at runtime in response to changing conditions. The VVM can be integrated with various types of distributed objects and middleware [32].

A simplified picture of the VVM architecture is shown in Figure 2, as applied to CORBA

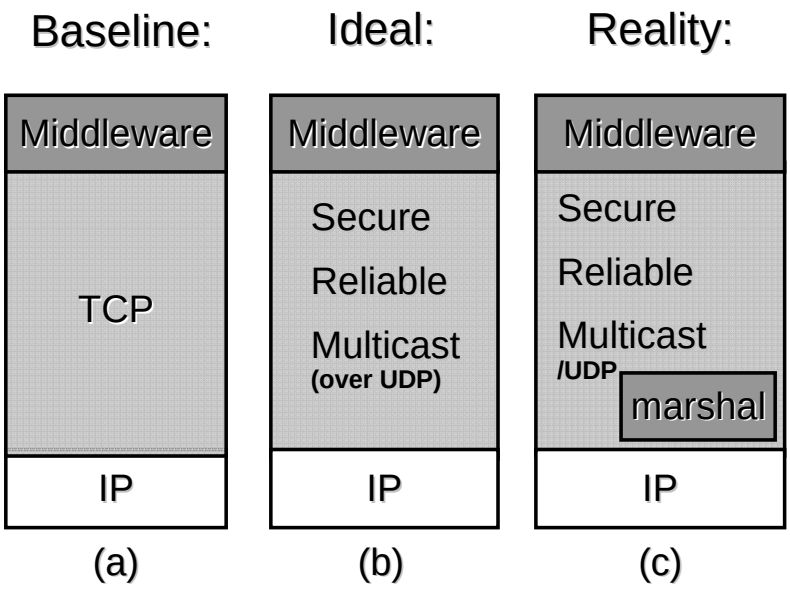


Figure 1: VVM Stack

(though it could equally well apply to any other middleware system).

The VVM operates in three general steps as follows.

- Step 1 Unmarshal: Network messages come in independently from each replica into the *unmarshal* module. This converts these linearized messages into a set of parameters, which are sent to the *voter core* module.
- Step 2 Vote: The *voter core* module selects/generates an "answer" (a set of parameters) based on the current voting policy.
- Step 3 Marshal: The *marshal* module gets the "answer" from *voter core* and "flattens" the voted parameters into one message, which will be sent to the server or client.

In this configuration the voter is external to any CORBA ORB — for example, in a gateway [33] — but it could also be embedded into an ORB, in which case the marshal and unmarshal operations of the ORB would be used, and the extra unmarshal/marshal steps for a gateway would be avoided.

The *voter core* module has inputs which inform it of the current voting policy (VDL code fragment) as well as notifying it when a failure occurs.

**NOTE: shall we include VSS in our paper?**If yes, add a short paragraph to describe the VSS.

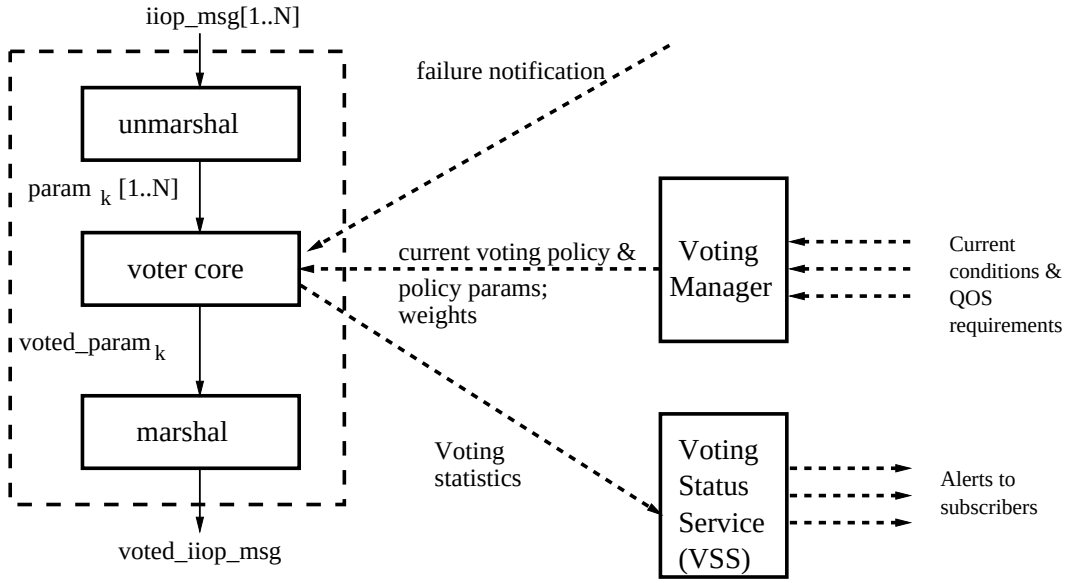


Figure 2: VVM Architecture

## 3.2 Basic Voter Functionality

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

The voter core shown in Figure 2 goes through three states in the processing of a single vote, as depicted in Figure 3.2.

Basicly, the *voter core* runs at one of the three states, *quorum*, *exclusion* and *collation*. At each state, the current voting policy dictates the action of the voter by providing operations, supplying necessary parameters and specifying conditions.

- At *quorum* state, the voter waits for enough ballots to arrive to begin the actual voting process. After sufficient ballots have arrived, the *exclusion* state is entered.
- At *exclusion* state, the voter excludes a number of parameters from further consideration, if necessary, to enable the application to tolerate value failures.
- At *collation* state, one value is chosen from those not excluded.

After the collation state is finished, the voter also may return a *confidence value*, indicating the level of confidence in the vote.

In each of the three states exceptions may be thrown as directed by the VDL. In this case,

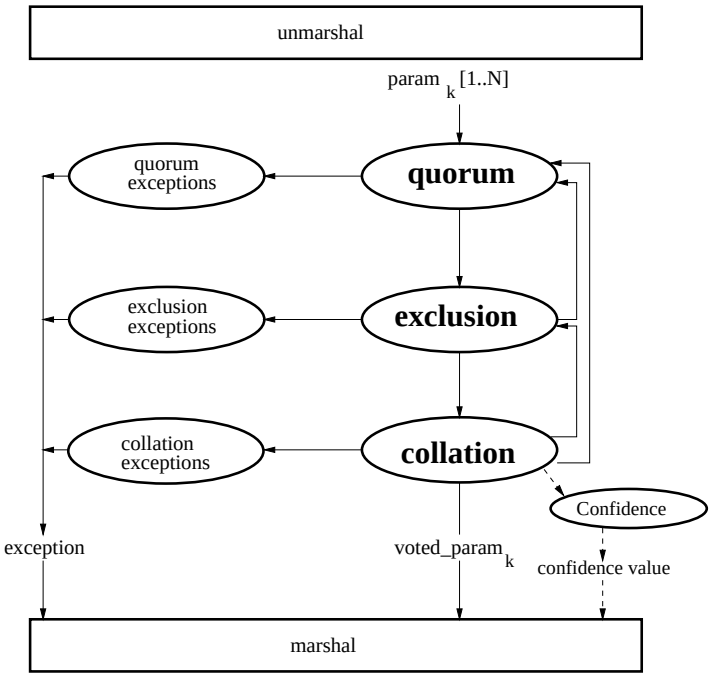


Figure 3: States in the Voter Core

an exception is returned to the client which 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.

**NOTE: this paragrph needs to be rewritten if branching is changed, i.e. branch from collation to exclusion be eliminated** In the last two states the voter may branch back to a previous state, based on the same status conditions on which an exception may be thrown. For example, in the exclusion state, if too many ballots are excluded, the voter can branch back to the quorum state to wait for more.

## 3.3 Advanced Voter Functionality

The voter can be specified to perform random operations at any or all of the three states, to help thwart an adversary; this is of course specified in VDL. For example, in *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 *exclusion* state the voter can exclude a random number of the ballots; again, this percentage could potentially be reset at different time

intervals. Finally, the collation state can randomly choose a value. This is well reasonable in some applications where the particularly bad data have already been discarded in the exclusion state.

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 towards 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. For example, suppose there are ballots with values $\{5,7,9\}$ from replicas 1–3, respectively, and the weighting is $\{2,1,3\}$. After the expansion the ballots will be $\{5,5,7,9,9,9\}$, and this is what the collation operation such as median or random will operate on.

## 3.4 Failure Model

**DAVEDO: Dave will rewrite this section. The following are copied from DSN paper**

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

The VVM also assumes a Byzantine failure model for applications running on servers, but only a crash failure model for the middleware and operating system code running on them. 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 [34] or with more expensive protocols similar to those in [35] or in [36].

```
policy name (parameters) {
    quorum (quorum_op (parameters))
        [ throw ex_name if (condition) ]*
    [ exclusion [label] (exclusion_op (parameters)) [ replace by mean|median|value]
        [
      [ throw exception_name if (condition) ]   |
      [ goto quorum [ (using (quorum_op (parameters)) more|total) ] if (condition) ]
        ]*
    ]+
    collation (collation_op (parameters))
        [
      [ throw exception_name if (condition) ]   |
      [ goto quorum [ (using (quorum_op (parameters)) more|total) ] if (condition) ]   |
      [ goto exclusion [label] if (condition) ]
        ]*
    [confidence (confidence_expression) ]
}
```

Figure 4: VDL Syntax

# 4 Voting Definition Language (VDL)

In this section[1] we describe VDL. we first give its syntax, then describe the primitive operations supported in the language. Finally we discuss confidence values.

## 4.1 VDL Syntax

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

A VDL policy begins with optional subtypes, which can be declared to restrict values of parameters to a policy, similar to the **range** statement in Ada. Subtypes can be applied to CORBA IDL **long** and **double**, to restrict their values, as well as to builtin **enum** types QuorumOps, ExclusionOps, and CollationOps that enumerate the operation names for each state in the voter. These are shown by example in Section 5.2. The next line declares a policy's name and parameters, which can include **long**, **double**, the builtin enumerated types for operations in each state.

Following this is the declaration of the **quorum** operation and its parameters. It can optionally be followed by a statement indicating that a named exception should be thrown to

[1] Please note that this research is not a theoretical investigation of how to represent all voting algorithms. Rather, it is a pragmatic investigation into how to support voting in middleware, though supporting as wide as set of algorithms as possible is obviously useful.

| CONDITION VARIABLES | MEANING |
|---|---|
| **elapsed_time** | the time since vote initialized (msec) |
| **pct_remaining_total** | the percentage of remaining ballots and total ballots |
| **num_remaining_total** | the total number of remaining ballots |
| **pct_excluded_total** | the percentage of excluded ballots and total ballots |
| **num_excluded_total** | the total number of excluded ballots |
| **ballots_total** | total # of ballots received, equals (**num_excluded_total** + **num_remaining_total** ) |
| **ballots_max** | total # of ballots possible, e.g. number of replicas; if > **ballots_total**, **quorum** does not wait for all ballots to arrive. |

Table 1: VDL Condition Variables

the client under certain conditions. These conditions are expressions built with the policy's parameters and also *condition variables* that the voter core provides regarding the status of a vote. These are listed in Table 1. As an example, **elapsed_time** denotes the milliseconds since the voter was initialized for a given vote, for example when a client's request was sent out. It is useful for example to set a timeout in the **quorum** state, or for confidence values.

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

Note that there may be multiple **exclusion** statements in a row. If there are more than one, and they may be branched to with a **goto** statement, then they must have the optional label to enable this branching without ambiguity.

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

| STATE | PRIMITIVES | MEANING |
|---|---|---|
| **quorum** | **until** $k[\%]$ | wait until $k[\%]$ ballots arrive |
| | **all_but** $k[\%]$ | wait until all but $k[\%]$ ballots arrive |
| | **random** $n[\%]$ $m[\%]$ | wait for random number (ranged from $n[\%]$ to $m[\%]$) of ballots arrive |
| **exclusion** | **lowest** $n[\%]$ | exclude the lowest $n[\%]$ values |
| | **highest** $n[\%]$ | exclude the highest $n[\%]$ values |
| | **furthest** $n[\%]$ | exclude the furthest $n[\%]$ values from the median |
| | **distance** $e[\%]$ **from** [**mean**\|**median**\|$value$] | exclude all values not within $e[\%]$ distance from the position or value specified |
| | **outside_sigma** $x$ | exclude all values more than $x$ std. dev. from the mean |
| | **distance_neighbor** $d$ | exclude all values that are not within a given distance $d$ of their neighbor |
| | **distance_cluster** $d$ [**mean**\|**median**] | exclude all values that are not within $d$ distance of their neighbor, starting at position specified |
| | **cluster_support** $d$ $p$ [$c$] | Refer to Table 3 for details |
| | **inner** $k$ | Refer to Table 3 for details |
| | **nearest** $k$ | exclude the nearest $k$ values from median |
| | **random** $n[\%]$ | exclude random $n[\%]$ |
| | **none** | exclude none |
| **collation** | **median** | choose the median value |
| | **mean** | choose the mean value |
| | **mean_neighbor** | choose the closest value in a ballot from the mean value |
| | **midpoint** | choose the mean value of the highest one and the lowest one |
| | **midpoint_neighbor** | choose the closest value in a ballot from the midpoint |
| | **mode** | choose the mode value (the most common one) |
| | **random** | choose a random one |

Table 2: VDL Primitives

If an exception is thrown by the VVM, it is done so in whatever runtime exception the middleware system requires clients to catch, as middleware systems generally do — e.g., `SYSTEM_EXCEPTION` for CORBA – so as not to introduce new exceptions into the interface.

## 4.2 VDL Primitives

Table 2 shows the primitives for each state. Most are intuitive and self-explanatory, so for brevity's sake are not explained here. Primitives in the **exclusion** state are applied after the values have been sorted.

Table 3 provides more information about the VDL primitives that may not be fully intuitive to all readers. In this table, a set of example values is given, with those that would be excluded are shown in a bold font.

Operation **distance_neighbor** $d$ excludes all values that are not within a given distance $d$ from their neighbor, after the values are sorted. Note that this may result in multiple **clusters** of neighbors, where each cluster is formed from neighbors $d$ or less apart but the clusters are

| PRIMITIVES | MEANING | EXAMPLES |
|---|---|---|
| **distance_neighbor** $d$ | exclude all values that are not within a given distance $d$ of their neighbor | **exclusion distance_neighbor** 2<br>{**8**, 11, 12, 14, 14, 15, 17, 18, **22** } |
| **distance_cluster** $d$ | exclude all values that are not in a chain starting at median and where neighbors are within $d$ | **exclusion distance_cluster** 2<br>{**9**, **11**, 12, 14, 14, 15, **17**, **18**, **22** } |
| **cluster_support** $d$ $p$ $[c]$ | Form clusters (which may overlap) including values that are each *supported by* (are within distance $d$ of) at least $p$ other values, as well as the $p$ values that form the support.<br>If $c$ is specified, exclude all but at most $c$ clusters, otherwise merely exclude all values not in a cluster. | **exclusion cluster_support** 1 2<br>{**9**, **11**, **12**, 14, 14, 15, 17, 18, 19 }<br><br>**exclusion cluster_support** 1 2<br>{**9**, **11**, **12**, 14, 14, 15, **17**, **18**, **20** }<br><br>**exclusion cluster_support** 1 2 1<br>{**9**, **11**, **12**, 14, 14, 15, **17**, **18**, **19** } |
| **inner** $k$ | exclude the median plus $k-1$ values adjacent to the median, spaced as evenly as possible on each side of the median | **exclusion inner** 3<br>{9, 11, 12, **14**, **14**, **19**, 21, 22, 22 } |
| **nearest** $k$ | exclude the median and $k-1$ values nearest it | **exclusion nearest** 3<br>{9, 11, **12**, **14**, **14**, 19, 21, 22, 22 } |

Table 3: VDL Exclusion Primitives with Examples

spaced more than $d$ apart. Operation **distance_cluster** forms a single cluster starting at the median or mean (as specified), and extending as far out as the adjacent value is within the given distance. All values not in this cluster are excluded. Operation **cluster_support** forms possibly overlapping clusters, where each member of a cluster is *supported* by at least $p$ other values, meaning they are within $d$ of the value. If $c$ is specified, it denotes the maximum number of clusters that will be allowed; all unallowed clusters, plus all values not in a cluster, are excluded. If $c$ is not specified, then all clusters are allowed; only values not in a cluster are excluded. Operation **inner** forms a cluster starting with the median and including $k-1$ other values, being balanced as well as possible on both sides of the median. The values in this cluster are excluded, and the others are not. Finally, operation **nearest** forms a cluster with the median and the $k-1$ values nearest it. These are excluded, and the rest are retained.

## 4.3 Confidence Values

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

One use is to allow the client to decide how to use a reply based on how good it is per-

ceived to be. The confidence value is presently specified in VDL, as described in Section 4.1. Alternately, we have been investigating allowing its specification via a separate *confidence definition language*. This would allow pieces of code (confidence policies) to be written by a different person from the VDL programmer, and they also could potentially be reused with different VDL policies. No matter how its specified, the confidence value will allow the client to adapt with a much better granularity than the current boolean output behavior.

We are also investigating utilizing the confidence value for application-level adaptation while still providing voting and replication transparency. This is enabled by using delegates, which are proxies that are interposed between the application client and the middleware proxy (stub), and provide the same API as that proxy (the API of which the client has programmed to). In particular, we are investigating the use of delegates from the Quality Objects (QuO) framework [37]. QuO delegates are generated based on QuO's Structure Description Language (SDL) [38] plus the middleware's interface definition language (IDL) and the QuO contract's contract description language (CDL). SDL allows the specification of adaptation strategies which are above the middleware layer (CORBA or DCOM, for example) yet below (and generally transparent to) the client. We are developing and analyzing the effectivness of SDL adaptation strategies which utilize the confidence value returned[2] by the voter.

# 5 VDL Examples

## 5.1 Supermajority

Figure 5 gives an example of a supermajority voting policy. The policy returns a vote of 60% of the ballots are equal, where we set "equal" to **median** being within 1%. It throws an exception if this cannot be met. Otherwise a confidence value is returned that is equal to the percent of ballots that agreed with the value voted upon (chosen in **collation**).

This policy initially waits for 60% of the ballots to arrive. It then discards all which are not equal. At this point, if more than 40% have been excluded, then a supermajority of 60%

[2]When we say a voter "returns" the confidence value, note that it cannot return this value as a return value, because the method's IDL definition has fixed the method's signature. Alternate means of returning the value are being developed, analogous to a Unix system call "returning" ("on the side") an error report via the `errno` variable. One possibility in implementing the confidence value involves CORBA context variables; others will be explored.

```
policy Supermajority-example {
        quorum (until(60%))
                throw QUORUM_TIMEOUT if (elapsed_time > 1000)
        exclusion (distance (1%) from median)
                throw BAD_VALUES if (pct_excluded_total > 40)
                goto quorum ( using (until (1)) more) if (pct_remaining_total < 60)
        collation (median)
        confidence (pct_remaining_total/100.0)
}
```

Figure 5: Supermajority Voting Algorithm in VDL

cannot be achieved, so an exception is thrown. If this does not happen, but less than 60% remain (which all are equal), then it branches back to the **quorum** state to wait for more. If neither the branch nor the exception occur, then the supermajority has been met, and **median** is chosen.

## 5.2 Parameterized Supermajority

The supermajority example given in Section5.1 can be parameterized in that the hardcoded constants (e.g. 60% and 1% in Figure 5) in voting policy can be defined as parameters to the policy. Further, the operations that are hardcoded can also be parameterized. For example, to exclude **furthest** values may also be feasible in the supermajority example.

In order to support parameterized policy, VDL built-in types need to be introduced. VDL predefines enumeration types (Figure 6) for QuorumOps, ExclusionOps, and CollationOps, each contains the legal operations which can be performed in *quorum*,*exclusion*, and *collation* state. Note that different VDL primitives have different number of arguments. To make this amenable to clean parameterization, VDL also defines a structure, called VDL_params, to pass in multiple parameters into a policy. The parameters are passed in by the Voter Manager, along with the policy.

Figure 7 gives the VDL for a supermajority algorithm. It is different from the earlier algorithm in that the percent to define a supermajority is parameterized, as well as the operations for **exclusion** and **collation**.

```
enum QuorumOps    {until, all_but, random,...};
enum ExclusionOps {lowest, highest, ..., random, none};
enum CollationOps  {median, mean, mean_neighbor, mode, random};
enum VDL_param_type {is_double, is_long, is_absent};

//a param is either long or double; and can also have '%'
struct VDL_params {
        double p1_double; long p1_long; VDL_param_type p1_type; boolean p1_pct;
        double p2_double; long p2_long; VDL_param_type p2_type; boolean p2_pct;
        double p3_double; long p3_long; VDL_param_type p3_type; boolean p3_pct;
};
```

Figure 6: VDL Built In Types for Parameterization

```
subtype Supermajority_exclusion_ops: ExclusionOps { lowest, furthest };
subtype Supermajority_quorum_increment: long [2, 4]; //how many more to wait for
policy    Supermajority( double pct_same,
                         Supermajority_quorum_increment q_inc,
                         Supermajority_exclusion_ops    ex_op,
                         VDL_params                     ex_params,
                         CollationOps                   c_op ) {
          quorum (until(pct_same %))
                    throw QUORUM_TIMEOUT if (elapsed_time > 1000) // 1000 msec
          exclusion (ex_op (ex_params))
                    throw BAD_VALUES if (pct_excluded_total > (100.0 - pct_same))
                    goto quorum using (until (q_inc) more) if (pct_remaining_total < pct_same)
          collation (c_op)
          confidence (pct_remaining_total/100.0)
}
```

Figure 7: Parameterized Supermajority Voting Algorithm in VDL

```
policy CNVA (d, local_value) {
        quorum (all)
                throw QUORUM_TIMEOUT if (elapsed_time > 1000)
        exclusion (distance (d) from mean) replace by local_value
                throw NOTHING_LEFT if (pct_excluded_total = 100.0)
        collation (mean)
        confidence (pct_remaining_total/100.0)
}
```

Figure 8: CNVA Clock Synchronization Algorithm in VDL

## 5.3 Fault-Tolerant Clocks

Fault tolerant clock is a very important topic in distributed computing area. People have proposed several different approaches, which can be categorized (in [39]) as master-slave, convergence function, interval, byzantine agreement, and clock server algorithms.

In this paper, we demonstrate the use of VDL to describe convergence function algorithms. A typical algorithm in this category contains four basic steps [39]:

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

Among those steps, the first three can be mapped onto our three voter states (discussed in Section 3.2): *quorum*, *exclusion* and *collation*. The last step is to employ a correction algorithm to adjust the local clock, which would be the application's function.

Four major convergence algorithms, Interactive Convergence, Fast Convergence, Fault-Tolerant Midpoint, and Fault-Tolerant Average Algorithms will be expressed in this paper. They have been identified as the most important convergence algorithms by many researchers [39–44].

Figure 8 gives an example of Interactive Convergence Algorithm (CNVA) [45]. The synchronizing node first collects clock values from all other nodes, and then it replaces the values that are "too far away" from $local\ value$, where "too far away" means not within a given distance $d$ from $local\ value$. If all the values are excluded (replaced), it throws an exception. After that, it calculates the corrected local clock value by choosing the **mean** of those manipulated values. Finally, the **confidence**

```
policy FCA (d, p) {
        quorum (all)
                throw QUORUM_TIMEOUT if (elapsed_time > 1000)
        exclusion (cluster_support (d, p))
                throw NOTHING_LEFT if (pct_excluded_total = 100.0)
        collation (mean)
        confidence (pct_remaining_total/100.0)
}
```

Figure 9: FCA Clock Synchronization Algorithm in VDL

```
policy FTMA (k) {
        quorum (all)
                throw QUORUM_TIMEOUT if (elapsed_time > 1000)
        exclusion (lowest (k))
        exclusion (highest (k))
                throw NOTHING_LEFT if (pct_excluded_total = 100.0)
        collation (midpoint)
}
```

Figure 10: FTMA Clock Synchronization Algorithms in VDL

value is computed according to how many values originally collected are excluded ($pct\_remaining\_total$).

Figure 9 gives an example of the Fast Convergence Algorithm (FCA) [46]. After collecting values from all other nodes, the synchronizing node excludes those values which receives support from less than $p$ other values, where $value_a$ "supports" $value_b$ means $|value_a - value_b| <= d$. If all the values are excluded, it throws an exception. After that, it calculates the corrected local clock value by choosing the **mean** of those manipulated values. Finally, the **confidence** value is computed similarly as in CNVA.

Figure 10 gives an example of the Fault-Tolerant Midpoint Algorithm (FTMA) [47]. After value collection (**quorum**), the synchronizing node nodes excludes the **lowest** $k$ and **highest** $k$ values, followed by excluding all the remaining values except the lowest and highest ones, i.e. the **inner** values are further excluded. If all the values are excluded, it throws an exception[3]. After that, it calculates the corrected local clock value by choosing the **mean** of those two remaining values. There is no **confidence** value computed in this case, there are only two values that are collated, and hence

[3]Note that, unlike the CNVA and FCA algorithms, the FTMA algorithm will exclude a number of ballots that does not depend on the values involved, but rather only on the number of ballots and $k$. Thus, the exception here is to catch a programmer error.

```
policy FTAA (k) {
        quorum (all)
                throw QUORUM_TIMEOUT if (elapsed_time > 1000)
        exclusion (lowest (k))
        exclusion (highest (k))
                throw NOTHING_LEFT if (pct_excluded_total = 100.0)
        collation (mean)
        confidence (pct_remaining_total/100.0)
}
```

Figure 11: FTAA Clock Synchronization Algorithm in VDL

there is no obvious generic **confidence** function, though many are possible.

Figure 11 gives an example of Fault-Tolerant Average Algorithm (FTAA) [43]. This algorithm is similar to FTMA except that instead of further excluding inner values and only using two remaining ones to compute the mean, the FTAA computes the mean using all the remaining values without further exclusion. The **confidence** value of FTAA is computed according to how many values originally collected are excluded.

# 6 Implementation

## 6.1 VVM System Elements

Figure 12 shows the elements of VVM system. In this figure, single lines denote runtime interactions between system elements; dashed lines denote the actual voting process, which is described in Section 3.2. There are two major components in VVM, Voter Core and Voter Manager. Together with other system specific components, i.e. WPMessageHelper (which stands for Wire Protocol Message Helper) and IDL Compiling Tools, they consist a general, portable, and adpatable voting middleware system. The Policy object connects the Voter Core and the Voter Manager. It has to be portable, which will be further discussed in Section 6.2.

The VSS (Voter Status Service) and IR2LUT (Interface Repository to Lookup Table) components are implemented in a previous version of VVM. The description can be found in [30] and will not be discussed further in this paper.

Please note that Figure 12 does not distinguish between local message passing and remote message tranmission. This is because the VoterCore and Client/ClientProxy

(Server/ServerProxy) can be either on the same or different machines.

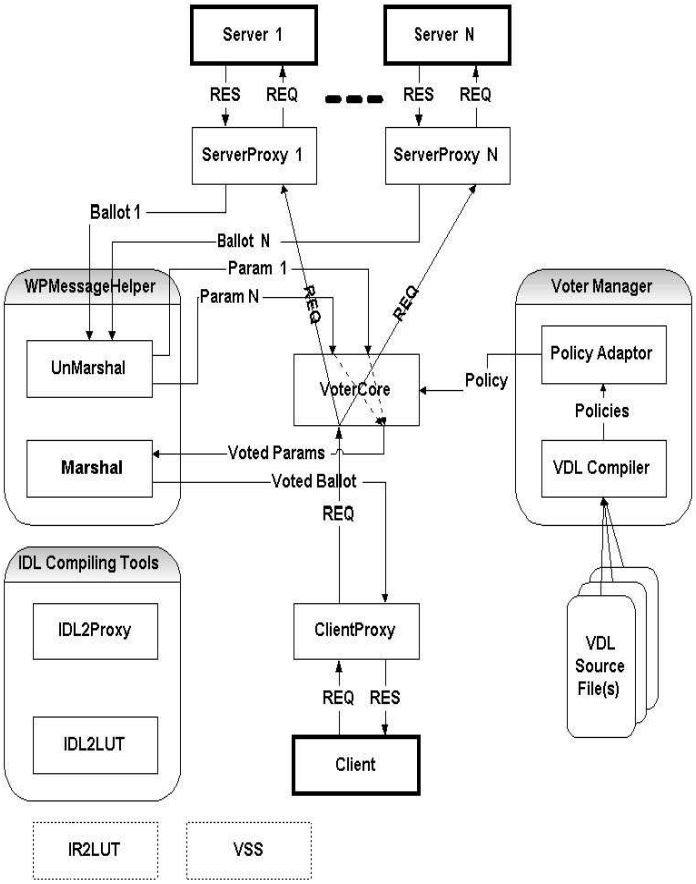


Figure 12: VVM 2.0 System Architecture and Main Message Flow

## 6.2 Portable Policy Structure

Adaptive voting requires the behavior of Voter Core be controled by the Voter Manager at runtime. In the VVM system, the adaptive control information is represented by a a Policy object, which is generated from a piece of VDL code by the VDL Compiler. Figure 13 gives the internal structure of such an object. Policy Header contains the descriptional information, i.e. the table length and version number. The rest of the Policy object consists of a list of tables.

One important feature of the Policy object is portability. Since the manager and core can be running on separate machines, it is critical that the core interprets the voting policy the same way as the manager wants it to. In Figure 13, there are seven tables , which altogether serve as a portable representation of a voting algorithm. The

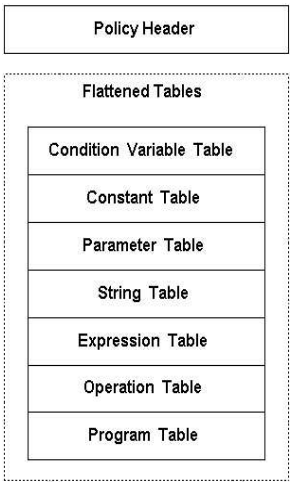


Figure 13: Policy Structure

first four tables, i.e., Condition Variable Table, Constant Table, Parameter Table and String Table, are the Data Tables. They contain the data that are used by other tables. The three remaining tables, i.e., Expression Table, Operation Table and Program Table, are the Control Tables. They contain the instructions, representing the voting policy. One simple voting policy example can be found in Figure 15. The according Data Tables and Control Tables can be found in Figure 16 and Figure 17 respectively. A detailed description on the format of each table is given in [31].

```
policy simple_policy_0( long a ) {
    quorum (until( 60% ))
        throw TIMEOUT if (elapsed_time > 500)
    exclusion (lowest( a ))
        goto quorum using (until (10%) more) if (pct_excluded_total > 10)
    collation (mean)
    confidence (pct_remaining_total / 100.0)
}
```

Figure 14: A Simple Voting Policy in VDL

## 6.3 Voter Manager, Voter Core and Adaptive Voting

The Voter Manager consists of two main modules: the Policy Adaptor and the VDL Compiler. The Policy Adaptor adaptively changes the Policy of the Voter Core, while the VDL Compiler parses and tranform a VDL file into a Policy object. The Voter Core

```
policy simple_policy_0 (long a) {
        quorum (until(60%))
                throw TIMEOUT if (elapsed_time > 500)
        exclusion lowest (a)
                goto quorum ( using (until (10%)) more) if (pct_excluded_total > 10)
        collation (mean)
        confidence (pct_remaining_total/100.0)
}
```

Figure 15: A Simple Voting Policy in VDL

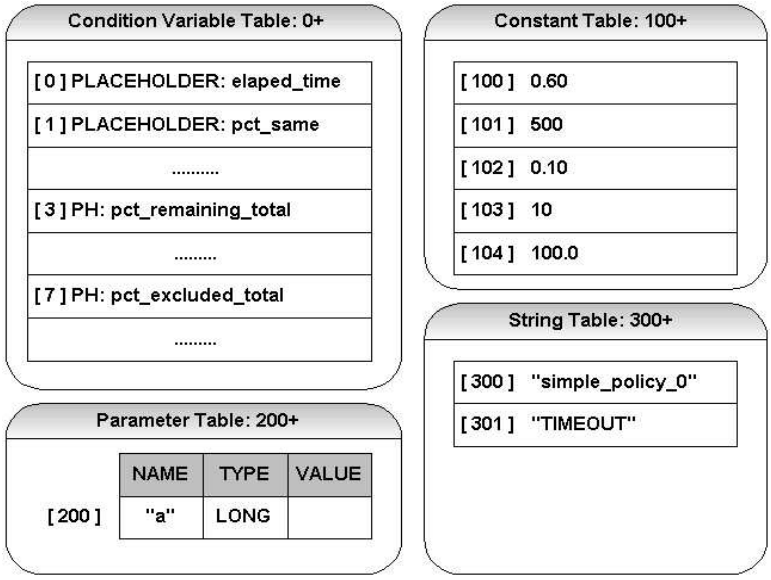


Figure 16: A Simple Voting Policy Structure (1) - Data Tables

| Condition Variable Table | VALUE |
|---|---|
| [0] | (value of elapsed_time) |
| [1] | (value of pct_remaining_total) |
| [2] | (value of num_remaining_total) |
| [3] | (value of pct_excluded_total) |
| [4] | (value of num_excluded_total) |
| [5] | (value of ballots_total) |
| [6] | (value of ballots_max) |

| Constant Table | VALUE |
|---|---|
| [100] | 0.50 |
| [101] | 600.0 |
| [102] | 0.10 |
| [103] | 10.0 |
| [104] | 100.0 |

| Parameter Table | NAME | TYPE | VALUE |
|---|---|---|---|
| [200] | "a" | LONG | (value filled in at run time) |

| String Table | VALUE |
|---|---|
| [300] | "simple_policy_0" |
| [301] | "TIMEOUT" |

Table 4: A Simple Voting Policy Structure (1) - Data Tables (w/ default size of each table equals 100)

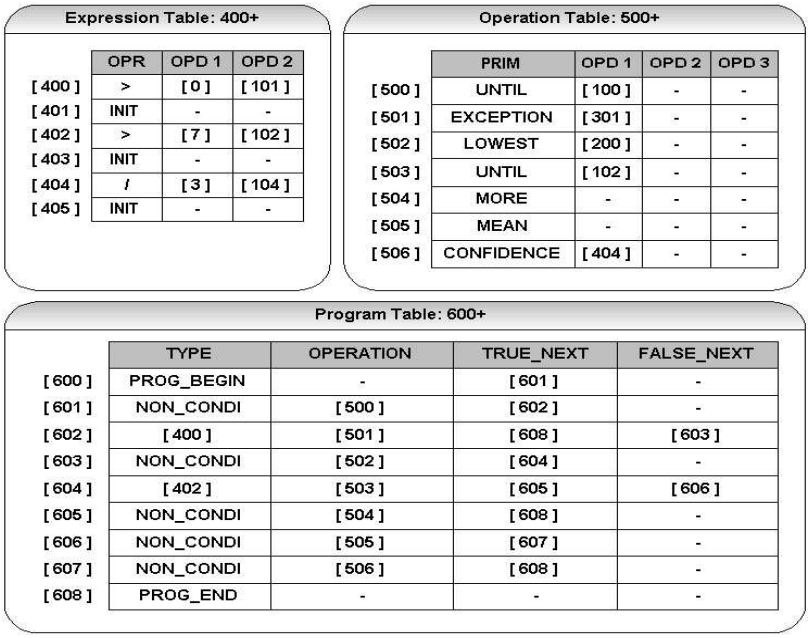


Figure 17: A Simple Voting Policy Structure (2) - Control Tables

| Expression Table | OPERATION | OPRAND1 | OPRAND2 |
|---|---|---|---|
| [400] | > | [0] | [101] |
| [401] | - | - | - |
| [402] | > | [7] | [102] |
| [403] | - | - | - |
| [404] | / | [3] | [104] |
| [405] | - | - | - |

| Operation Table | PRIMITIVE | OPRAND1 | OPRAND2 | OPRAND3 |
|---|---|---|---|---|
| [500] | UNTIL | [100] | - | - |
| [501] | EXCEPTION | [301] | - | - |
| [502] | LOWEST | [200] | - | - |
| [503] | UNTIL | [102] | - | - |
| [504] | MORE | - | - | - |
| [505] | MEAN | - | - | - |
| [506] | CONFIDENCE | [404] | - | - |

| Program Table | OPERATION_TYPE | OPERATION | TRUE_NEXT | FALSE_NEXT |
|---|---|---|---|---|
| [600] | PROG_BEGIN | - | [601] | - |
| [601] | NON_CONDI | [500] | [602] | - |
| [602] | [400] | [501] | [608] | [603] |
| [603] | NON_CONDI | [502] | [604] | - |
| [604] | [402] | [503] | [605] | [606] |
| [605] | NON_CONDI | [504] | [608] | - |
| [606] | NON_CONDI | [505] | [607] | - |
| [607] | NON_CONDI | [506] | [608] | - |
| [608] | PROG_END | - | - | - |

Table 5: A Simple Voting Policy Structure (2) - Control Tables (w/ default size of each table equals 100)

also consists of two main modules: the Policy Interpreter and the Vote Information Table. The Policy Interpreter interprets the Policy and performs the actual voting process, while the Vote Information Table maintains the storage of all the unmarshaled ballots, history and voting results.

Adaptive voting in VVM is implemented by the Voter Manager updating the Policy object of the Voter Core at runtime. The manager has both interactive and automatic policy updating interfaces. The user of the system can force the manager to update a new policy by using the interactive updating interface. He/she can also specify a list of system conditions for the manager to watch. Once those conditions are satisfied, the manager will update the policy through automatic interface, without the interaction with user.

In the current VVM system, the manager uses **push** to update the policy. When the manager tries to update the policy, the core may have a vote which is currently being processed. For example, the voter core may wait in the quorum state for some vote when a new Policy object is received. It will be ambiguous if the policy of the current vote be updated. Therefore, the new policy can only affect the votes that "happen" after it is received at the core.

## 6.4 VVM Development Environment

The VVM Development Environment is the a set of standalone tools used to facilitate the VVM system. It currently contains the WPMessageHelper and the IDL Compiling Tools.

The WPMessageHelper is an assistant component to the Voter Core. It contains the Wire Protocol Message format information and does the **Unmarshal/Marshal** work. The VVM system allows application specific wire protocol to be used to transfer ballots. Therefore, to have a separate WPMessageHelper in the VVM instead of integrating it in the core is important. In some application settings, the Voter Core may reside in the CORBA ORB. In this case, the WPMessageHelper is bypassed and the Unmarshal/Marshal in the CORBA ORB will be used.

In order to unmarshal/marshal the ballots, the WPMessageHelper has to know the

function signature for correctly interpreting parameter/return types. A lookup table constaining the interface name, function name, direction (REQUEST/REPLY), and type signatures is generated for this purpose by IDL2LUT (IDL to Look-Up Table) tool, which is part of the IDL Compiling Tools. A simple IDL to lookup table mapping example is given in Figure 18.

```
interface  grad_student{
        float doing_thesis( inout long  pages,in float  time );
        void  doing_project( out long  lines );
};

interface  professor{
        void enjoying_vacation( inout float  time );
        long pizza_meeting( inout long  pizzano, out float  calories);
};
```

| INTERFACE NAME | METHOD NAME | DIRECTION | PARAMETER LIST |
|---|---|---|---|
| grad_student | doing_thesis | request | { long, float} |
| grad_student | doing_thesis | reply | { float, long} |
| grad_student | doing_project | request | { } |
| grad_student | doing_project | reply | { void, long} |
| professor | enjoying_vacation | request | { float} |
| professor | enjoying_vacation | reply | { void, float} |
| professor | pizza_meeting | request | { long} |
| professor | pizza_meeting | reply | { long, long, float} |

Figure 18: IDL to Lookup Table Mapping

Please note that in current VVM system, only function definitions and three data types in IDL (void, long and float) are supported by IDL2LUT. More data types and IDL features will be supported in the future.

The CORBA Interface Repository provides dynamic, run-time access to interface metadata, which can be used to generate Lookup table. A separate tool, IR2LUT, will also be integrated in the VVM environment.

Another tool, IDL2PROXY (IDL to Proxy generator), is used to generate Client-Proxy/ServerProxy. It is also part of the IDL Compiling Tools.

```
interface grad_student {
        float doing_thesis(inout long pages, in float time);
        void doing_project(out long lines);
};
interface professor {
        void enjoying_vacation(inout float time);
        long pizza_meeting(inout long pizzano, out float calories);
};
```

| Interface Name | Method Name | Direction | Parameter List |
|---|---|---|---|
| grad_student | doing_thesis | request | {long, float} |
| grad_student | doing_thesis | reply | {float, long} |
| grad_student | doing_project | request | { } |
| grad_student | doing_project | reply | {void, long} |
| professor | enjoying_vacation | request | {float} |
| professor | enjoying_vacation | reply | {void,float} |
| professor | pizza_meeting | request | {long} |
| professor | pizza_meeting | reply | {long,long,float} |

Table 6: IDL to Lookup Table Mapping

# 7 Performance

**DAVEDO: Dave will provide this section**

Description: The performance measurement was done on a server machine equipped with 4 Intel Xeon processors (3.2GHz each) and 4GB RAM. It runs Red Hat Enterprise Linux AS release 4 (2.6.9-55.ELsmp). The voting policy used is *quorum until(100%) exclusion lowest (1) collation mean*. The results are the average of 10 runs. All the parameters are float point values.

"P" in Table 7 stands for the number of parameters to vote on. "S" in Table 7 and Figure 19 stands for the number of server replicas. The "Unmarshal" time measured in Table 7 is the total time of unmarshalling all the ballots, while the "Process" time meaured in the same table is the time of **exclusion** and **collation**. The "Total" time measured in Figure 19 is the total time of unmarshal, process and marshal time in Table 7. In other words, it is the actual time spent by the VoterCore on computing, not including the time spent on waiting for enough ballots to come (**quorum** time).

# 8 Discussion

**DAVEDO: Dave will provide part of this section**

| - | - | P=1 | P=2 | P=3 | P=4 | P=5 |
|---|---|---|---|---|---|---|
| S=3 | Unmarshal | 14.9 | 14.6 | 15.3 | 16.0 | 15.8 |
| | Process | 10.9 | 13.2 | 15.5 | 17.8 | 20.0 |
| | Marshal | 2.1 | 2.1 | 2.7 | 2.1 | 1.9 |
| S=5 | Unmarshal | 22.3 | 22.8 | 24.6 | 24.2 | 23.8 |
| | Process | 12.2 | 15.2 | 18.7 | 20.5 | 29.9 |
| | Marshal | 1.8 | 2.1 | 2.2 | 2.2 | 2.3 |
| S=7 | Unmarshal | 21.6 | 28.5 | 30.4 | 29.1 | 32.5 |
| | Process | 12.8 | 17.1 | 19.6 | 22.1 | 24.6 |
| | Marshal | 2.1 | 2.0 | 2.0 | 2.5 | 1.9 |
| S=9 | Unmarshal | 37.9 | 36.2 | 38.0 | 37.6 | 38.9 |
| | Process | 13.2 | 15.9 | 20.6 | 24.2 | 29.3 |
| | Marshal | 1.7 | 1.7 | 1.8 | 2.1 | 2.0 |
| S=11 | Unmarshal | 46.1 | 46.1 | 48.1 | 43.6 | 49.0 |
| | Process | 13.8 | 17.6 | 21.7 | 25.1 | 28.2 |
| | Marshal | 1.9 | 1.9 | 2.1 | 2.1 | 2.1 |
| S=13 | Unmarshal | 53.9 | 52.7 | 51.3 | 53.8 | 51.9 |
| | Process | 14.3 | 18.9 | 22.6 | 26.6 | 31.1 |
| | Marshal | 2.1 | 2.0 | 2.0 | 2.2 | 2.0 |
| S=15 | Unmarshal | 56.5 | 53.0 | 57.8 | 57.5 | 55.9 |
| | Process | 14.8 | 21.9 | 26.0 | 30.8 | 34.7 |
| | Marshal | 2.0 | 2.0 | 2.5 | 2.0 | 2.5 |

Table 7: VoterCore Categorized Running Time (unit: $10^{-6}$ second)

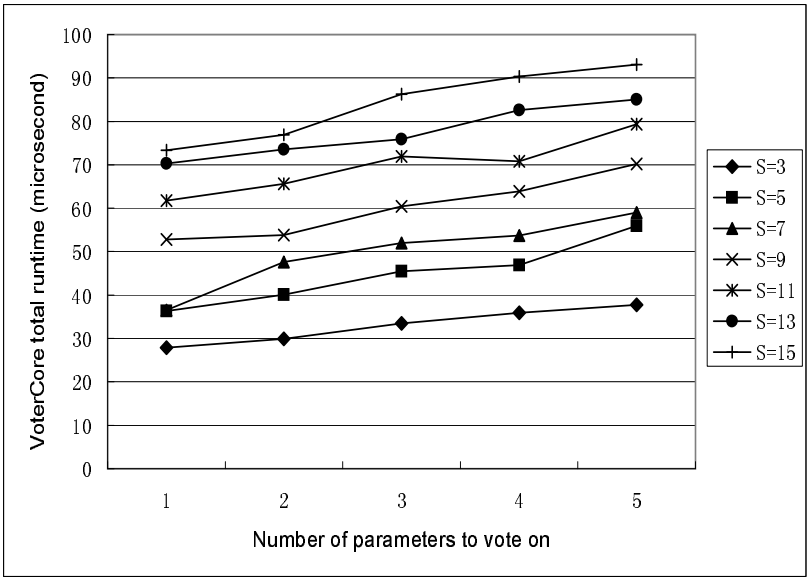


Figure 19: VoterCore Total Running Time

## 8.1 VDL and VVM versus Java for Voting Policies

**DAVEDO: Dave will provide this section**

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

Less delicately, while VDL may (or may not) be considered clever, is it just something that late night talk show host David Letterman might call "stupid middlware tricks?"

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

## 8.2 Branching and "Goto" Primitive

Branching happens when the internal state of voter core chang Exclusion to Quorum, or Collation to Exclusion, or Collation to Quorum. As described in Section 4 and Section 3.2, branching is important for voting optimization. Under the assumption that the system will run under normal condition (nothing "bad" happens) in most of the time, branching can save a lot of time when performing voting.

In current VVM system and VDL, branching is implemented by "goto" primitive in both exclusion and collation. However, one of the lesson we learn from the development of VVM is that this solution really adds much complexity to the Policy structure, VDL compiler and policy interpreter. Firstly, in order to support the "goto" statement, we have to keep a Java Bytecode like format of Policy structure. A much simpler structure, which just contains a sequential list of the primitives that are used in each state, cannot be applied. Secondly, the VDL compiler has to spend extra time

on deciding the destination of the "goto" statement and filling it in the Policy object. Finally, the policy interpreter has to retain extra history information in order to support runtime "goto".

Moreover, "goto" primitive can cause potential "deadloop" problem if the VDL programmer is not careful enough, which adds the complexity of voting algorithm design.

We are now exploring using function calls inside voting policy to replace the "goto" primitive.

## 8.3 Voting on Non-Basic Types

**NOTE: copied from TR**

So far the VDL presented allows voting on basic types such as based on integer or floating point numbers. In some cases, however, it is useful to use the VVM to vote on another basic type or on a user-defined abstract data type (ADT). These could represent, for example, an application structure, an object's state, or a string. We also can use the same mechanism to vote on exceptions, which in many middleware frameworks contain an ADT, though this requires some simple modifications to the VVM which we have planned.

While it is in some cases desirable to vote on one, it is not inherently obvious how to vote on an arbitrary ADT, or even one that it is given the declaration of. For example, even given a description of its fields, there are many design choices that could reasonably be made multiple ways, and some ADTs would have no obvious ways to vote on.

However, it is still useful to be able to vote on them. To enable this, the VVM allows programmers to tell it of ADTs as well as how they should be handled. The developer must define the ADT in a CORBA IDL **struct**, then provide helper objects for one of three ways we allow it to tell how to vote on the ADT; each involves implementing a Java **interface** (recall the VVM is currently implemented in Java, though this could be generalized to a CORBA interface).

The first way is fairly simple: the helper object maps between the ADT and a

CORBA **double**. Clearly not all ADTs will have an intuitive mapping between them and doubles, but for those that do this provides a very simple way to use their ADT with the VVM. The second way is implementing Java `comparable` interface for a child class of the ADT — which alows for testing of lesser, equal, or greater — plus providing method bodies for the few operations such as **mean** and **mean_neighbor** (which are described below) that do not make sense by comparing alone. The third way is to implement a method for all of the primitives that the VVM supports.

Besides providing the helper object for the voter core by one of these three techniques, the developer must also provide a helper object to marshal and unmarshall the **struct**; we later plan on automating this with an IDL compliler. Finally, the developer must fill in a configuration table noting the ADT's interface name, which one of the three porting techniques was chosen, and the classes of the helper objects.

## 8.4 Multi-parameter Voting

The VVM system and VDL presented in this paper also support **multi-parameter voting**. In multi-parameter voting, each ballot contains more than one datum, and all of them will participate in the voting process. In real life, multi-parameter voting is extremely useful in building data fusion applications, where there is a high chance that the server may returns more than one single value at a time. Multi-parameter voting can also be used to implement voting on non-basic data types. For example, it is well suitable to use multi-parameter voting technique to implement voting on a complex structure, if it consists of only a list of basic data types.

There are two basic problems of supporting multi-parameter voting. The first one is how to exclude "bad"[4] values. There are two approaches which can be applied here. The first one is call the "value" based approach, which means if a "bad" value is detected, only this value is excluded, other values in the same ballot will still be kept. Another approach is the "ballot" based approach, which means if a value in a ballots is detected to be "bad", the whole ballot will be excluded. This approach may be more suitable to fault-tolerant system designers in that if something goes wrong, more

[4]By "bad", we mean those values that are specified in VDL to be excluded

likely the things that are coming together with it are wrong too. Both approaches can be implemented on the same system structure. Currently the VVM system supports the "value" based approach.

Another problem of supporting multi-parameter voting is how to use policies to control the voting process. One simple approach is to use a single policy to control the voting on all types of data, which is currently supported by VVM. This approach simplifies making adaptive decision in the Voter Manager but lacks flexibility. Another approach is to specify different policy for each data type. For example, use "policy_float" to vote on all the float point values and use "policy_integer" to vote on all the integer values. We define **voting pattern** to denote such association between policies and data types. We are currently exploring formalization of this type of association. More discussions and examples on multi-parameter voting are given in [31].

# 9 Related Work

**NOTE: Copied from TR version of DSN paper. Some of the stuff should be cut in order to save space**

## 9.1 Synchronization 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 [48]. It was generalized to weighted voting in [49], where different replicas are given a different number of votes. There have been a number of generalizations of voting such as dynamic voting [50], multi-dimensional voting [51], and voting with witnesses [52].

A recent and scalable example is Phalanx [53].

BBN's Cronus (CORBA-like) middleware [54–56] has had replication with synchronization voting support since the mid-1980s; Arjuna is a more recent system using such mechanisms [57–59]. 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.

## 9.2 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 [60]. The earliest example of collation voting is the SIFT (Software-Implemented Fault Tolerance) project at SRI [61]. SIFT was designed for real-time aircraft control, and featured eight processors running in loose synchrony on the order of 50 $\mu$sec. SIFT's application software is structured as a set of iterative tasks that are run at a frequency that depends 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 [62, 63]. 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 [35,64], using Secure Agreement Protocols [65]. 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 [66, 67], Eternal [68], AQuA [69]. These systems all have the virtue of providing a high degree of (but not total) application transparency and also of being useful with commercial middleware. However, they all have a very limited form of voting: either 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 marshaled parameter buffer. The Immune system provides survivability to CORBA applications via active replication, voting, and a secure multicast protocol [70]. Like Rampart [35, 64], it thus covers a wider range of malicious faults than does the VVM. However, there are no details on the voting design or implementation other than it votes on both requests and replies, and waits for a majority "being identical in value;" We thus presume it employs byte-by-byte voting on the marshalled parameter buffer.

## 9.3 N-Version Programming

Active replication (which uses synchronization voting) has been developed to mask hardware failures of nodes and communication links. N-Version programming is a technique used to mask software design faults [71, 72]. In this approach, N versions of a module are independently developed from the same specification and executed on separate (typically heterogeneous) nodes. The return value or "answer" from each is collated using collation voting.

We note that N-Version programming suffers from the **consistent comparison problem** [73], which limits its use in error detection due to the finite precision in computer arithmetic. Here, two replicas can compute the same variable with slightly

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

## 9.4 Voting Algorithms

Theoretical work on voting dates back more than fifteen years. Section 5 provides details of some of the major work in this field, including various clock synchronization and other algorithms which VDL can express.

There are some algorithms which VDL and VVM cannot presently express, though when we extend VDL to cover voting on multiple parameters we hope to cover many of these. These include distance agreement protocols and distance decision [74], which involve voting in multi-dimensional space; the generalization of commonly used voting techniques such as Formalized Majority Voter, Generalized Median Voter, etc. in [75]; and the adjudicators and adjudication functions [76], which are generalized concepts of voters and voting algorithms.

There are also several software systems that are well known for their implementation works of voting. One of them is the UCLA DEDIX [77] system, which is a distributed testbed for multiple-version software. Another one is the fault-tolerant avionics application of algorithm diversity described in [78].

Please note the research work described in this thesis is not a theoretical attempt

to provide a generalized voting service or notation that strives to express all possible voting algorithms. Rather, it is a pragmatic effort to create an embeddable middleware voting component. Certainly we have tried to make VDL be as general as possible — and it covers much of the space of voting algorithms in the literature, as discussed in [30] — but covering the entire space is not one of its goals.

# 10 Conclusions

**DAVEDO: Dave will provide this section**

# References


[1] T. Anderson, M. Dahlin, J. Neefe, D. Pat-terson, D. Roselli, and R. Wang, "Serverless network file systems," in *In Proceedings of the 15th Symposium on Operating System Principles. ACM*, (Copper Mountain Resort, Colorado), pp. 109–126, December 1995.

[2] M. N. Nelson, Y. A. Khalidi, and P. W. Madany, "The Spring file system," Tech. Rep. SMLI TR–93–10, 1993.

[3] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun Network Filesystem," in *Proc. Summer 1985 USENIX Conf.*, (Portland OR (USA)), pp. 119–130, 1985.

[4] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[5] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," in *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[6] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2000.

[7] J. R. Douceur and R. P. Wattenhofer, "Large-scale simulation of replica placement algorithms for a serverless distributed file system," in *9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2001.

[8] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," in *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[9] J. Gwertzman and M. I. Seltzer, "World wide web cache consistency," in *USENIX Annual Technical Conference*, pp. 141–152, 1996.

[10] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, "A hierarchical internet object cache," in *USENIX Annual Technical Conference*, pp. 153–164, 1996.

[11] P. Cao and C. Liu, "Maintaining strong cache consistency in the world wide web," *IEEE Transactions on Computers*, vol. 47, no. 4, pp. 445–457, 1998.

[12] OMG, "The omg homepage: http://www.omg.org," Feb 26 2006.

[13] CORBA, "The corba homepage: http://www.corba.com," Dec 2 2006.

[14] D. McKinnon, O. Haugan, T. Damania, K. Dorow, W. Lawrence, and D. Bakken, "A configurable middleware framework with multiple quality of service properties for small embedded systems," in *Technical Report, WSU, TR-2002-37*, 2002.

[15] P. Narasimhan, L. Moser, and P. Melliar-Smith, "State synchronization and recovery for strongly consistent replicated corba objects," in *Proceedings of the IEEE Int. Conf. on Dependable Systems and Networks (DSN)*, 2001.

[16] A. Arora, G. Leon, and S. Wallace, "The corba replication service," As in Feb 26, 2006.

[17] J. Adamec, M. Grf, J. Kleindienst, F. Plsil, and P. Turingma, "Supporting interoperability in corba via object services," Feb 26 2006.

[18] B. Liskov, M. Castro, L. Shrira, and A. Adya, "Providing persistent objects in distributed systems," *Lecture Notes in Computer Science*, vol. 1628, 1999.

[19] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. G. ord, and M. F. Kaashoek, "Rover: A toolkit for mobile information access," in *the 15th ACM Symposium on Operating Systems Principles*, pp. 156–171, 1995.

[20] D. B. Terry, M. M. Theimer, K. Peterson, A. J.Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the 15th ACM SOSP*, 1995.

[21] Kazza, "The kazza homepage: http://www.kazza.com," Nov 12 2006.

[22] Napster, "The napster homepage: http://www.napster.com," Nov 12 2006.

[23] Gnutella, "The gnutella homepage: http://www.gnutella.com," Nov 12 2006.

[24] J. Kangasharju, K. Ross, and D. Turner, "Secure and resilient peer-to-peer e-mail: Design and implementation," in *Proceedings of IEEE International Conference on P2P Computing, 2003.*, 2003.

[25] Clip2, "The gnutella protocol specification v0.4 document revision 1.2," *http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf*.

[26] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," *Lecture Notes in Computer Science*, vol. 2009, pp. 46+, 2001.

[27] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz, "Maintenance-free global data storage," *IEEE Internet Computing 5(5), 40-49.*, 2001.

[28] A. I. T. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *Symposium on Operating Systems Principles (SOSP)*, pp. 188–201, 2001.

[29] D. Bakken, D. Karr, C. Jones, and J. Hale, "The voting virtual machine: A flexible mechanism for collating replicated client requests and server replies," in *FTCS-29 FastAbstract Proceedings, IEEE*, (Madison, WI), June 1999.

[30] C. C. Jones, "The voting virtual machine: Voting support for distributed systems," Master's thesis, School of Electrical Engineering and Computer Science, Washington State University, May 2000.

[31] Z. Zhan, "Adaptive voting and data fusion in middleware," Master's thesis, School of Electrical Engineering and Computer Science, Washington State University, June 2001.

[32] D. E. Bakken, Z. Zhan, and C. C. J. andDavid A. Karr, "Middleware support for voting and data fusion," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN2001)*, IEEE, June 2001.

[33] R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquier, and J. Joyall, "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.

[34] 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.

[35] 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.).

[36] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ACM, February 1999.

[37] 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.

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

[39] M. J. Pfluegl and D. M. Blough, "A new and improved algorithm for fault-tolerant clock synchronization," *Journal of Parallel and Distributed Computing*, vol. 27, pp. 1–14, 1995.

[40] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems," *IEEE Trans. Comput.*, vol. C-36, pp. 933–940, August 1987.

[41] P. Ramanathan, K. Shin, and R. Butler, "Fault-tolerant clock synchronization in distributed systems," *Computer*, vol. 23, pp. 33–42, October 1990.

[42] P. Ramanathan, D. Kandlur, and K. Shin, "Hardware-assisted software clock synchronization for homogeneous distributed systems," *IEEE Trans. Comput.*, vol. 39, pp. 514–524, April 1990.

[43] J. Lundelius-Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Inform. and Comput.*, vol. 77, pp. 1–36, April 1988.

[44] P. Thambidurai, A. Finn, R. Kieckhafer, and C. Walter, "Clock synchronization in MAFT," *Digest of 19th International Fault-Tolerant Computing Symposium*, pp. 142–149, 1989.

[45] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *J. Assoc. Comput. Mach.*, vol. 21, pp. 52–78, January 1985.

[46] S. Mahaney and F. Schneider, "Inexact agreement: Accuracy, precision, and graceful degradation," in *Proc. 4th ACM SIGACT-SIGOPS Symposium, Principles in Distributed Computing*, pp. 237–249, 1985.

[47] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl, "Reaching approximate agreement in the presence of faults," *J. Assoc. Comput. Mach.*, vol. 33, pp. 499–516, June 1986.

[48] 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.

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

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

[51] 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.

[52] 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.

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

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

[55] 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.

[56] 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.

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

[58] 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.

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

[60] 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.

[61] 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.

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

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

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

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

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

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

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

[73] 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.

[74] K. Echtle, "Distance agreement protocols," in *Proceedings of the Nineteenth Annual International Symposium on Fault-Tolerant Computing*, pp. 191–198, 1989.

[75] P. R. Lorczak, A. K. Caglayan, and D. E. Eckhardt, "A theoretical investigation of generalized voters for redundant systems," in *Proc. 19th IEEE Int. Symp. on Fault-Tolerant Computing*, pp. 444–451, 1989.

[76] F. D. Giandomenico and L. Strigini, "Adjudicators for diverse-redundant components," in *Proc. IEEE 9th Symposium on Reliable Distributed Systems*, pp. 114–123, 1990.

[77] A. Avizienis, P. Gunningberg, J. Kelly, L. Strigini, P. Traverse, K. Tso, and U. Voges, "The UCLA DEDIX system: A distributed testbed for multiple-version software," *Digest of 15th Annual International Symposium on Fault-Tolerant Computing*, pp. 126–134, June 1985.

[78] A. K. Caglayan, P. R. Lorczak, and D. E. Eckhardt, “An experimental investigation of software diversity in a fault-tolerant avionics application,” in *Proceedings of Symposium on Reliable Distributed Systems*, pp. 63–70, 1988.

# Contents

# List of Figures

# List of Tables