The Voting Virtual Machine: A Flexible Mechanism for Collating Replicated Client Requests and Server Replies

David E. Bakken, David A.Karr
Distributed Systems Department
BBN Technologies
Cambridge, Mass. USA
{dbakken,dkarr}@bbn.com

Christopher C. Jones, John C. Hale
School of Elec. Eng. and Computer Science
Washington State University
Pullman, Wash. USA
{cjones,hale}@eecs.wsu.edu

1. Introduction

In any configuration with actively replicated server objects, it is necessary to collate replicated replies into one to return to the client. Additionally, when the clients themselves may be replicated, as with a nested invocation chain, the requests the client replicas send out must be collated into a single request to be sent to each of the server replicas.

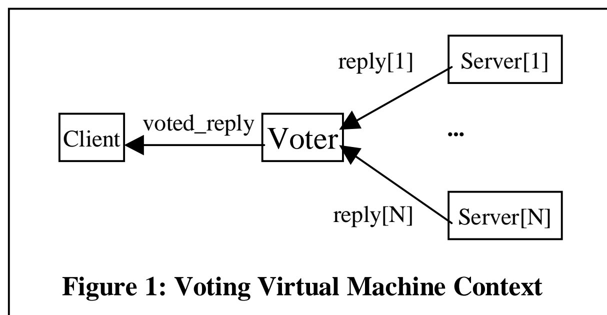

Figure 1: Voting Virtual Machine Context

Figure 1 shows a simple example of this. Each reply arrives at the voter, which must choose one and send it on to the client.

Current support for voting is quite limited, however, in both research projects and commercial products and standards [1,2,3,4]. Such voting support generally allows at best the specification of "majority" or "all", based only on the return value of a method. These schemes implicitly assume that all methods have a return value and that no values from returned (**out** or **inout**) parameters will be used, and they are inflexible when return values are floating point values. Further, they are undefined where the clients are replicated and thus multiple requests must be collated into one.

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 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.)

In this paper we introduce the Voting Virtual Machine (VVM), which purposes to make such voting much more flexible, and outline the issues our research will be covering in the near future.

2. VVM overview

We have designed and are implementing the Voting Virtual Machine (VVM), and its companion Voting Description Language (VDL) policy language, to overcome these limitations. It allows VDL Policies which can be used for multiple kinds of method signatures, as well as ones more specific to a given method. It will work with multiple presentation layers such as those for CORBA and DCOM, as well as with our simple (but structurally similar) test presentation layer. It allows voting policies to be changed at runtime. It supports both *static voting*, where the voting algorithms cannot be influenced by group membership changes, and *dynamic 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, security, and survivability concerns.

Figure 2 gives a simplified picture of the VVM architecture. Messages ("flattened" for transmission, e.g., into CORBA GIOP's CDR format, which is used with IIOP) come in independently from each replica into the *unmarshal* module. This converts the linearized message into a vector of parameters (e.g., {*e*,*f*} for a reply to *method2*). The vectors of parameters are sent to the *voter* module, which selects a set of parameters to be returned to the client. Finally, the *marshal* module

© 1999, David E. Bakken

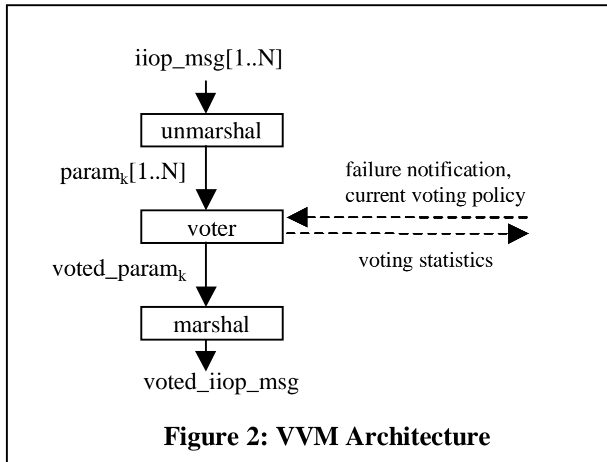

Figure 2: VVM Architecture

"flattens" the voted parameters into one message, which will be sent to the client.

The *voter* module has inputs which 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 to help detect performance problems, intrusions, or other anomalies.

3. Voter states and their VDL primitives

The voter goes through three states in the process of voting on a single vector of parameters. At each state, the VDL code for the current policy dictates the action of the voter in each state.

The first state is *quorum*, where the voter is waiting until enough messages have come in to begin voting. VDL primitives allow a fixed "*k* messages", or dynamic voting directives such as "all but *k", "50%",* etc.

After the quorum state's criterion has been met, the voter proceeds to the *exclusion* state. Here, some of the messages may be excluded, to make it more difficult for an adversary to inject bad data into the voted parameters. VDL primitives which specify the actions at the exclusion state can exclude "lowest *k*" values, "highest *k*" values, "*k* random" values, "all values outside *x* sigma of the mean value", etc.

The remaining parameter vectors are passed onto the *collation* state, where these vectors are combined into one. VDL primitives allow the collation to be specified as "mode", "median", "mean", "random", etc.

VDL allows for the specification of exceptions to be thrown to the client under a number of circumstatnces; e.g., timeout in the quorum state.

4. Open Issues

This research is just starting to scratch the surface in what we think will be a very fruitful avenue of research. As such, there are still many outstanding questions. We outline a few of these briefly.

- **Replicated voters**: how must VDL be constrained when the voters are actively replicated, especially if we wish the voter replicas not to have to synchronize to make consistent decisions?
- **Floating point support**: what considerations must be accounted for when the parameters can be floating point, especially if there can be multiple CPU architectures involved?
- **Multiple parameter collation**: what possible ways make sense 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. However, for some, we believe it will be useful to choose the "best" parameter for each parameter in a reply message (e.g., the best value of *e* and *f* in *method2*'s reply above), then combine these "best" values into one reply message. Note that in this case the actual vector selected may not have been generated by any server. For example, if Server[1] returned {1,2} and Server[2] returned {3,4}, we might create a reply message of {3,2} by choosing the "best" of each parameter (where the VDL specifies what is best).
- **Weighted voting**: how can we employ weighted voting to give the replies from different servers unequal weight in the voting? This can help improve security or survivability, presumably, by allowing more trusted domains to have a greater influence on the vote.
- **Multiple vote statistics**: status information beyond a single vote may be useful indicators of a number of anomalies. E.g., where a given replica is often or always the last to reply, or where its parameters are always far from the mean.
- **Arrays**: How can we combine replicas of an array parameter, perhaps with different lengths? Similar issues apply to other compound data structures.

References

[1] S. Landis and S. Maffeis, "Building Reliable Distributed Systems with CORBA", *Theory and Practice of Object Systems*, 3(1), 1997, 31-43.

[2] L. Moser and P. M. Melliar-Smith and P. Narasimhan, "Consistent Object Replication in the Eternal System", *Theory and Practice of Object Systems*, 4(2), 1998. See also their FTCS-29 paper.

[3] http://www.omg.org/techprocess/meetings/schedule/Fault_Tolerance_RFP.html.

[4] M. Cukier *et al*, "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects", in SRDS-98 proceedings.