AN ADAPTIVE ALGORITHM FOR TOLERATING VALUE FAULTS AND CRASH FAILURES [1]

Jennifer Ren, Michel Cukier, and William H. Sanders

Center for Reliable and High-Performance Computing

Coordinated Science Laboratory and Department of Electrical and Computer Engineering

University of Illinois at Urbana-Champaign

Urbana, Illinois 61801

{ren, cukier, whs}@crhc.uiuc.edu

ABSTRACT

The AQuA architecture provides adaptive fault tolerance to CORBA applications by replicating objects and providing a high-level method that an application can use to specify its desired level of dependability. This paper presents the algorithms that AQuA uses, when an application's dependability requirements can change at runtime, to tolerate both value faults in applications and crash failures simultaneously. In particular, we provide an active replication communication scheme that maintains data consistency among replicas, detects crash failures, collates the messages generated by replicated objects, and delivers the result of each vote. We also present an adaptive majority voting algorithm that enables the correct ongoing vote while both the number of replicas and the majority size dynamically change. Together, these two algorithms form the basis of the mechanism for tolerating and recovering from value faults and crash failures in AQuA.

Keywords: Dependable distributed systems, replication protocols, adaptive fault tolerance, CORBA, group communication systems.

1. INTRODUCTION

Building fault-tolerant distributed applications is an important but difficult task. Recently, increasing attention has been given to developing middleware, intended to run on COTS, to provide fault tolerance to applications. The same middleware can be used for multiple, possibly changing, applications at once, and since different applications may have different fault tolerance needs, the middleware should provide adaptive fault tolerance. The AQuA architecture [Cuk98, Sab99] is one such approach to building dependable distributed systems. In particular, the AQuA architecture provides adaptive fault toler-

[1] This research has been supported by DARPA Contracts F30602-96-C-0315, F30602-97-C-0276, and F30602-98-C-0187.

ance to CORBA applications by replicating objects, has a high-level method (using Quality Objects (QuO) [Zin97, Loy98]) that applications use to specify their desired dependability levels, and has a dependability manager that attempts to reconfigure a system at runtime so that dependability requests are satisfied. AQuA uses the Maestro/Ensemble [Hay98, Vay98] group communication system to provide reliable multicast and total ordering. Previous papers on AQuA have addressed the overall architecture of AQuA [Cuk98, Ren99] and the algorithms used to tolerate crash failures [Sab99]. In this paper, we focus on the issue of simultaneously tolerating value faults[2] in applications and crash failures when an application's dependability requirements can change at runtime.

Majority voting using a group of replicated entities is a well-known technique for tolerating value faults in a distributed system. Generally speaking, such algorithms collate messages generated by the replicated entities into groups (one for each request/reply), vote on the messages, and then send out each majority value. In addition, through comparison of the messages received by each replica in the group, faulty replicas (which sent incorrect values) can be detected. If one assumes that the size of a replicated object group and the number of value faults to tolerate are constant, the resulting majority voting algorithm is relatively simple.

Unfortunately, neither of these values is constant for systems providing adaptive fault tolerance. In particular, voting schemes for practical dependable distributed systems should provide automatic detection of and recovery from value faults, even when replicas leave the group and new replicas are created and join the group during voting. The replicas may leave because of crash failures or because they are killed due to the detection of a value fault; they may join or leave the group if an application dynamically changes its required dependability, thus precipitating a change in the required number of replicas. Furthermore, when the number of value faults to tolerate is changed by an application, the corresponding majority size for the voting must be changed, in addition to the number of replicas. Each of these situations must be accounted for in a practical scheme for simultaneously tolerating and recovering from value faults and crash failures.

This paper makes two contributions. First, it provides an active replication communication scheme that tolerates value faults in applications and crash failures by maintaining data consistency among a group of replicas, detecting crash failures, collating messages generated by the replicated objects, and delivering the result of each vote. Second, it introduces an adaptive majority voting algorithm that pro-

[2] A *value fault* is a fault in which, for a "given service item," "the value of" the service item "does not fall within the set of values specified" for the service item [Pow92].

vides the majority value to the objects that should receive requests/replies and detects value faults, even when the number of replicas that participate in voting and the majority size dynamically change at runtime. Together, these two algorithms form the basis of the mechanism for tolerating value faults and crash failures in AQuA. (Note that our aim is not to tolerate value faults in the group communication system itself. If tolerance of more complex fault types is required, one could substitute a more secure underlying group communication protocol, like SecureRing or Rampart, in place of Ensemble within the AQuA architecture.)

The remainder of the paper is organized as follows. Section 2 provides a short review of the AQuA architecture, and describes how group communication is used to provide reliable multicast in AQuA. Section 3 describes an active replication communication scheme for making reliable remote method invocations, and for tolerating crash failures and value faults. Section 4 first illustrates the difficulties of majority voting when the group membership and the majority size change. It then presents an adaptive majority voting algorithm that makes it possible for the ongoing vote to be performed correctly even in the presence of changes in the number of voting replicas and the dependability level requested. Section 5 describes the implementation of the presented algorithms. Several related research projects are described and compared to AQuA in Section 6. Section 7 concludes the paper.

2. AQUA REVIEW

Figure 1 shows the various components of the AQuA architecture and their interactions. The AQuA system uses the Ensemble group communication system [Hay98, Bir96] to ensure reliable communication between groups of processes. Ensemble assumes that the process failures are fail-silent (i.e., fail only by crashing), and detects process failures through the use of “I am alive” messages. The AQuA architecture uses this detection mechanism to detect crash failures. Maestro [Vay98] provides an object-oriented interface (in C++) to Ensemble. In order to provide a simple way for application objects to specify and adapt the levels of dependability they desire, the AQuA architecture uses the *Quality Objects* (QuO) [Zin97, Loy98] framework to process and invoke dependability requests. QuO allows distributed applications to specify QoS requirements at the application level using the notion of a “contract.” Note that the AQuA architecture intentionally provides dependability to a distributed application in a semi-transparent way, allowing an application writer the freedom to change dependability requirements at runtime without concerning himself or herself with the details of the underlying mechanisms that provide fault tolerance.

Proteus, implemented on top of Maestro/Ensemble, is a flexible infrastructure for providing adaptive fault tolerance. Proteus makes remote objects dependable by using 1) a replicated dependability manager, 2) object factories, and 3) handlers in gateways. The *Proteus dependability manager* makes decisions regarding reconfiguration based on reported faults and dependability requests from QuO, and, together with the gateways, implements the chosen fault tolerance approach. Depending on the choices made by the dependability manager, Proteus can tolerate and recover from crash failures and value faults in application objects and QuO. *Object factories* are used to kill and start replicated applications and to provide information regarding the host to the dependability manager.

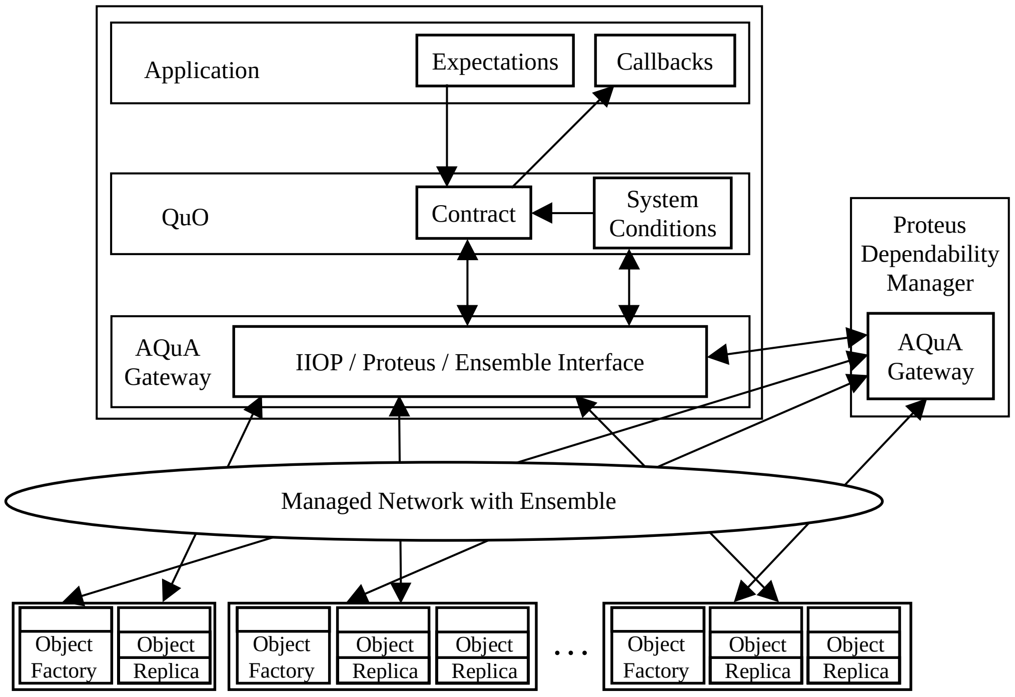


Figure 1. Overview of the AQuA Architecture

AQuA gateways provide applications with standard CORBA interfaces by translating between process-level communication, as supported by Ensemble, and IIOP messages, which are understood by Object Request Brokers (ORBs) in CORBA. CORBA provides application developers with a standard interface for building distributed object-oriented applications, but currently available commercial ORBs do not provide a simple approach that allows applications to be fault tolerant[3]. By using AQuA gateways, CORBA-based distributed applications that use the AQuA architecture can use standard, commercially

[3] Note that a CORBA fault tolerance standard has recently been specified by the OMG (see [OMG99]), but commercial products implementing the standard have not been released.

available ORBs. In addition to providing basic reliable communication services for application objects and QuO, gateways provide fault tolerance using different *voters* and *replication protocols*. These functions are located in the *gateway handlers.* A handler is created in the gateway for each pair of replicated objects that wish to communicate. Currently, six types of handlers are supported in the gateway: four replication handlers, the dependability manager handler, and the factory handler. Different handler types are used depending on the type of group in which the object is communicating. More specifically, the four replication handlers are an active replication handler with a pass-first policy (described in [Sab99]), an active replication handler with a leader only policy (a communication scheme, close to semi-active replication, in which only the leader sends out its request), an active replication handler with a majority voting policy (described in this paper), and a passive replication handler (in which the state is transferred after each output message) [Rub00].

The AQuA architecture uses group communication to provide reliable multicast. The basic unit of replication is the "AQuA object." An AQuA object is either a two-process pair (including an application and a gateway) or a three-process pair (consisting of an application, QuO, and a gateway). QuO is included if an object contained in the application process makes a remote invocation of another object and wishes to specify a quality of service for that object. When we say that "an object joins a group" we mean that the gateway process of the object joins the group. Mechanisms are provided to ensure that if one of the processes in the object crashes, the others are killed, thus allowing us to consider the object as a single entity that we want to make dependable. Four types of groups are used in the AQuA architecture: *replication groups*, *connection groups*, the *PCS* (*Proteus Communication Service*) *group*, and *point-to-point groups*. By defining multiple small replication and connection groups, we can avoid the communication overhead that would be incurred if a single large group were used.

A *replication group* is composed of one or more replicas of an AQuA object. A replication group has one object that is designated as its leader and may perform special functions. Each object in the group has the capacity to become the object group leader, and a protocol is provided to ensure that a new leader is elected when the current leader fails. For implementation simplicity, the object whose gateway process is the Ensemble group leader is designated the leader of the replication group. This allows Proteus to use the Ensemble leader election service to elect a new leader if the object leader fails.

A *connection group* is a group consisting of the members of two replication groups that wish to communicate. A message is multicast within a connection group in order to send a message from one replication group to another replication group. The replication group sending the message must communicate according to the communication scheme specified by the connection group. In our implementation, there are two connection groups for each pair of replication groups, depending on the direction of the message. The *sender connection group* is used to multicast messages reliably from the *sender replication group* (which sends out the request) to the *receiver replication group* (which sends out the reply). The *receiver connection group* is used to multicast messages reliably from the receiver replication group to the sender replication group.

Reliable multicast to the dependability manager is achieved using the *Proteus Communication Service (PCS) group.* The PCS group consists of all the dependability manager replicas in the system. The PCS group also has transient members. These transient members are object factories, AQuA application objects, and QuO objects that want to multicast messages to the dependability manager replicas. Since these transient members seldom need to communicate in the PCS group and the time needed for them to join the group when they need to communicate is not significant, the choice of having transient members instead of permanent members increases the performance of the AQuA system. Through the PCS group, AQuA gateways provide notification of view changes, QuO makes requests for dependability, and object factories respond to start and kill commands and provide host information updates. After a multicast to the PCS group, the transient member will leave the group.

Finally, a *point-to-point group* is used to send messages from a dependability manager to an object factory.

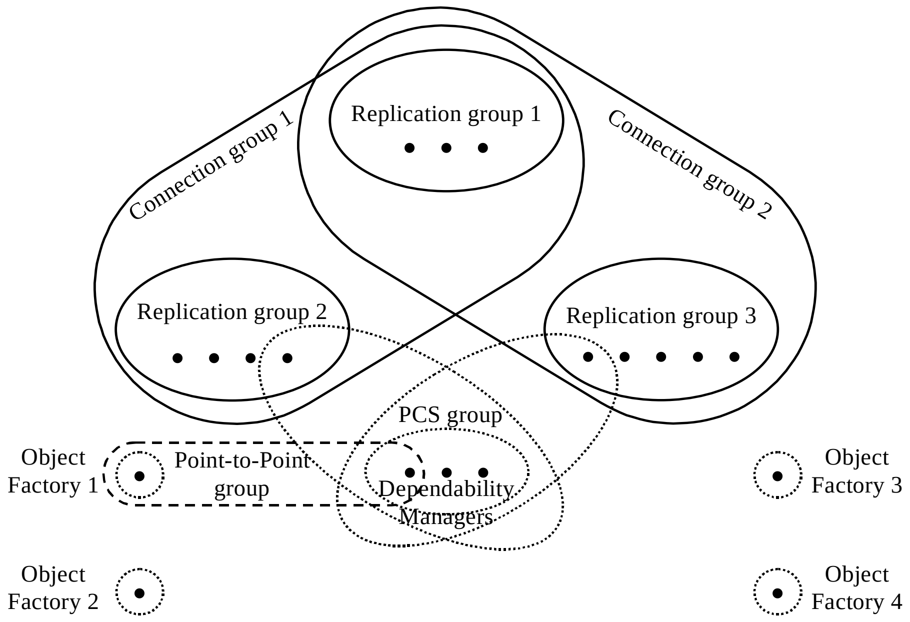


Figure 2. Example Group Structure in AQuA

For an illustration of replication groups, connection groups, the PCS group, and point-to-point groups, consider Figure 2. Solid lines define the replication and connection groups. Dashed lines represent different instances of the PCS group and the point-to-point groups. We see in Figure 2 that even though a connection group is composed of two replication groups, a replication group can be included in several connection groups. The structure of the PCS group in the figure shows that the leaders of replication groups 2 and 3 are communicating with the dependability manager.

3. COMMUNICATION SCHEME FOR TOLERATING VALUE FAULTS AND CRASH FAILURES

This section presents a communication scheme for simultaneously tolerating value faults in applications and crash failures when an application's dependability requirements can change at runtime.

3.1. Communication Scheme

In order to tolerate both value faults and crash failures, three steps are used to transmit a request or reply from one replicated object to another replicated object. We briefly describe these steps here, before providing detailed algorithms in the next subsection. In particular, assume that a replication group *i* generates a request and wishes to send it to replication group *j*. In the first communication step, each replica in group *i* multicasts the request in replication group *i* (step 1 in Figure 3). The leader records each request in a buffer, and when a majority of requests are received with the same value, forwards the request to replication group *j* by multicasting it in the connection group (step 2 in Figure 3). The other (non-leader) replicas record the request in a buffer, but do not forward the request to the remote replication group. (The buffer corresponding to a request is deleted when non-leaders see that the request has been multicast in the connection group.) By recording these requests in buffers, all non-leaders record enough information that any one of them can become the leader if the leader fails, and continue the voting and forwarding operation without losing any requests.

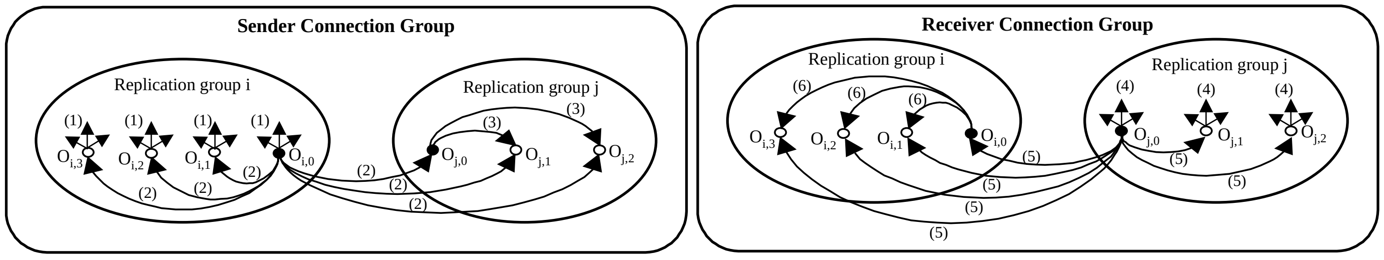


Figure 3. Communication Scheme

The third step in the scheme is necessary because Ensemble ensures that the messages are totally ordered within the same group, but it does not ensure that messages are totally ordered among different groups. This presents a problem, since there might be multiple replication groups that wish to send requests to group *j* via different connection groups; as a result, different replicas in group *j* might see different message orders. In order to ensure that all replicas of the destination group see requests in the same order, the leader of the receiver group multicasts the message in the destination replication group (step 3 in Figure 3). The gateway then forwards each request to the application, which processes the request. After the request is processed, the reply is transmitted back to the requestor using three steps (step 4, step 5, and step 6) that are similar to the steps just described for transmitting the request to the destination replication group.

3.2. Communication Scheme Algorithms

This section describes in detail the steps employed by the value/crash communication scheme in the order in which they are executed as a single request/reply is processed, using the "step" nomenclature introduced in the previous section.

To explain these steps, we first need to describe the structure of internal messages transmitted between gateways, and the state information maintained by gateways in order to process requests/replies correctly. Specifically, message headers contain *Sender* (replica or dependability manager), *Receiver* (replica or dependability manager), *SequenceNumber, Opcode, ID*, and *Endian* fields. It is important to note that the terms "sender" and "receiver," as used here, refer to the sender and receiver of the CORBA *request* that is made, not the sender or receiver of the message during a particular transmission. The sequence number is a positive number associated with the request/reply and is unique when tagged with both group names. The opcode is used to distinguish the message's purpose at a given step in the communication scheme. The ID contains the name of the host sending the request. The "endianess" of the message is also stored, and is used by the destination ORB. The relationship between opcodes and communication steps is shown in Table 1.

Step	Opcode	Step	Opcode
1	FORWARD_REQUEST	4	FORWARD_REPLY
2	CONNECTION_GROUP_REQUEST	5	CONNECTION_GROUP_REPLY
3	REPLICATION_GROUP_REQUEST	6	REPLICATION_GROUP_REPLY

Table 1. Communication Steps and their Opcodes

Two variables (*LastSent* and *LastDelivered*) are associated with each connection group, and keep track of the sequence numbers of the last sent and last delivered messages, respectively. The *LastSent* variable is used to ensure that a reply will not be dispatched to the application before the application has generated the corresponding request. Use of the *LastDelivered* variable ensures that no duplicate messages are delivered to an application object. There are also several buffers associated with each gateway (*ReplyBuffer, TotalOrderBuffer, MulticastDelayBuffer, MajorityBuffer, MajorityDelayBuffer,* and *ValueFaultBuffer*) that are used to enable data consistency among replicas and recovery from a leader crash failure. Each *ReplyBuffer* contains replies that have been received for requests that the replica has not yet sent out. Each *TotalOrderBuffer* contains requests (replies) multicast in the connection group that have not yet been multicast in the receiver (sender) replication group. Each *MulticastDelayBuffer* contains requests (replies) multicast in the receiver (sender) replication group that have not yet been received in the connection group. Each *MajorityDelayBuffer* contains majority values that have not yet been reached by the replica, but have already been reached by the leader and hence multicast in the connection group. Each *MajorityBuffer* contains the majority values that have been reached by the replica, but have not yet been multicast by the leader in the connection group. Finally, each *ValueFaultBuffer* contains the names of the replicas that have been judged by the associated gateway to be faulty. The precise use of these buffers will be explained when the communication steps are detailed.

Given this background, we can describe more precisely the sequence of steps that are used to transmit a request/reply from one replicated object to another and explain how reliable multicast and buffers are used to ensure that the remote object invocation is reliable, even if replicas crash and applications exhibit value faults.

Step 1: The first step begins with the multicasting of the request in the sender replication group. When **SendRequest** (shown in Figure 4) is called, each message is tagged with the opcode FORWARD_ REQUEST to indicate that it is in the first communication step, and with the hostname ID to identify which replica sent the request. *LastSent* for the sender connection group is then set to the sequence number of the message. Next, the reply buffer associated with the sender connection group is checked to see whether a reply has already been received for this request. The reply will be in the buffer if another replica has previously forwarded its request to the leader and has already received the reply. If the reply is present, it is delivered to the application. Regardless of whether the reply has been received, the request is multicast in the replication group, since it can be used to detect value faults.

```
SendRequest( message request )
    request.Opcode := FORWARD_REQUEST
    request.ID := HostName
    SenderConnectionGroup.LastSent := request.SequenceNumber
    message reply := RemoveReplyFromReplyBuffer( request )
    if ( reply ≠ NULL )
            DeliverReplyToApp( reply )
    MulticastToReplicationGroup( request )

SendReply( message reply )
```

```
reply.Opcode := FORWARD_REPLY
reply.ID := HostName
ReceiverConnectionGroup.LastSent := reply.SequenceNumber
MulticastToReplicationGroup( reply )
```

Figure 4. **SendRequest** and **SendReply** Methods in Value/Crash Communication Scheme

When a replica in the sender replication group receives the request multicast, **ReceiveReplicationGroupMulticast** (shown in Figure 5) is called. First, the origin of the message is checked. If the sender is the dependability manager and the IIOP message operation field associated with the message is *setVoterMajoritySize*, **ResetVoterMajoritySize** (described in Section 4.3.4) is called to change the majority size. If the majority size is decreased, it may cause some votes to reach their majorities, and hence become ready to be sent out to the remote replication group. This action is expressed in the lines following the **ResetVoterMajoritySize** invocation. Otherwise, since we are describing step 1, the opcode of the message will be FORWARD_REQUEST and will thus be changed to CONNECTION_GROUP_ REQUEST. The voter then calls **MajorityVotingProcess**, which adds the current message to the appropriate pool to be voted on and returns a set of zero or more requests on which a majority vote has been achieved. These requests are then multicast in connection groups associated with this replication group (details will be given in Section 4.3.2).

```
ReceiveReplicationGroupMulticast( message m )
    if ( m.Sender = DependabilityManager && m( operation ) = setVoterMajoritySize )
        messagequeue SetOfMessagesToMulticast := ResetVoterMajoritySize( m )
        While ( SetOfMessagesToMulticast ≠ NULL )
            MessageToMulticast := Dequeue( SetOfMessagesToMulticast )
            if ( Leader )
                MulticastToConnectionGroup( connectionGroup, MessageToMulticast )
            if (( NotLeader ) && ( RemoveMajorityDelayBuffer( MessageToMulticast ) = NULL ))
                AddToMajorityBuffer( MessageToMulticast )
    else
        if ( m.Opcode = FORWARD_REQUEST )
            m.Opcode := CONNECTION_GROUP_REQUEST
            messagequeue SetOfMessagesToMulticast := MajorityVotingProcess( m )
            While ( SetOfMessagesToMulticast ≠ NULL )
                MessageToMulticast := Dequeue( SetOfMessagesToMulticast )
                if ( Leader )
                    MulticastToConnectionGroup( connectionGroup, MessageToMulticast )
                if (( NotLeader ) && ( RemoveMajorityDelayBuffer( MessageToMulticast ) = NULL ))
                    AddToMajorityBuffer( MessageToMulticast )
            if ( Leader )
                if ( valueFaultBuffer ≠ NULL )
                    ReportValueFaultToDM( valueFaultBuffer )
                if ( noMajorityToReport ≠ NULL )
                    ReportNoMajorityToDM( noMajorityToReport )
        if ( m.Opcode = FORWARD_REPLY )
            m.Opcode := CONNECTION_GROUP_REPLY
            messagequeue SetOfMessagesToMulticast := MajorityVotingProcess( m )
            While ( SetOfMessagesToMulticast ≠ NULL )
                MessageToMulticast := Dequeue( SetOfMessagesToMulticast )
                if ( Leader )
                    MulticastToConnectionGroup( connectionGroup, MessageToMulticast )
                if (( NotLeader ) && ( RemoveMajorityDelayBuffer( MessageToMulticast ) = NULL ))
                    AddToMajorityBuffer( MessageToMulticast )
            if ( Leader )
                if ( valueFaultBuffer ≠ NULL )
                    ReportValueFaultToDM( valueFaultBuffer )
                if ( noMajorityToReport ≠ NULL )
                    ReportNoMajorityToDM( noMajorityToReport )
        if ( m.Opcode = REPLICATION_GROUP_REQUEST )
            if ( m.SequenceNumber > ReceiverConnectionGroup.LastDelivered )
                DeliverRequestToApp( m )
                ReceiverConnectionGroup.LastDelivered := m.SequenceNumber
                if ( RemoveFromTotalOrderBuffer( m ) = NULL )
                    AddToMulticastDelayBuffer( m )
        if ( m.Opcode = REPLICATION_GROUP_REPLY )
            if ( m.SequenceNumber > SenderConnectionGroup.LastDelivered )
                SenderConnectionGroup.LastDelivered := m.SequenceNumber
                if ( SenderConnectionGroup.LastSent ≥ m.SequenceNumber )
                    DeliverReplyToApp( m )
                else
                    AddToReplyBuffer( m )
                if ( RemoveFromTotalOrderBuffer( m ) = NULL )
                    AddToMulticastDelayBuffer( m )
```

Figure 5. **ReceiveReplicationGroupMulticast** Method in Value/Crash Communication Scheme

If the replica is the leader, it then multicasts each request in the appropriate connection group. If the replica is not the leader, and the replica has not yet seen the request multicast in the connection group by the leader (as determined by the outcome of the **RemoveMajorityDelayBuffer** call), the request will be stored in the *MajorityBuffer*. Storage in the *MajorityDelayBuffer* will occur if the leader has reached the majority value and multicast the request in the connection group before the replica itself has achieved the majority value for this request (see **ReceiveConnectionGroupMulticast**, Figure 6). Storing requests in the *MajorityBuffer* ensures that they will not be lost if the leader of the group crashes before it completes the voting process and multicasts them to the connection group.

```
ReceiveConnectionGroupMulticast( message m )
    if ( m.Sender = myReplicationGroup )
        if ( m.Opcode = CONNECTION_GROUP_REQUEST )
            if ( RemoveMajorityBuffer( m ) = NULL )
                AddToMajorityDelayBuffer( m )
        if ( m.Opcode = CONNECTION_GROUP_REPLY )
            m.Opcode = REPLICATION_GROUP_REPLY
            if ( RemoveMulticastDelayBuffer( m ) = NULL )
                AddToTotalOrderBuffer( m )
            if ( Leader )
                MulticastToReplicationGroup( m )
    if ( m.Receiver = myReplicationGroup )
        if ( m.Opcode = CONNECTION_GROUP_REQUEST )
            m.Opcode := REPLICATION_GROUP_REQUEST
            if ( RemoveMulticastDelayBuffer( m ) = NULL )
                AddToTotalOrderBuffer( m )
            if ( Leader )
                MulticastToReplicationGroup( m )
        if ( m.Opcode = CONNECTION_GROUP_REPLY )
            if ( RemoveMajorityBuffer( m ) = NULL)
                AddToMajorityDelayBuffer( m )
```

Figure 6. **ReceiveConnectionGroupMulticast** Method in Value/Crash Communication Scheme

Finally, if the replica is the leader, two checks related to voting are done. First, if the *valueFaultBuffer* is not empty (which is the case when value faults have been detected), the names of the faulty replicas held in the buffer are transmitted to the dependability manager. The procedure for placing items in this buffer is described in Section 4.3.2. Second, if the flag *noMajorityToReport* is true (which means that no majority has been reached for a vote, and all requests are in), that fact is reported to the dependability manager.

Step 2: The second communication step begins with the **MulticastToConnectionGroup** call. When the multicast message is received by a connection group member, **ReceiveConnectionGroupMulticast** is called (Figure 6). Since each member of the connection group receives the multicast, **ReceiveConnectionGroupMulticast** needs to determine whether the request came from a replica in the same or a different replication group. To distinguish between these cases, both the header and the opcode of the request are checked. In step 2, the opcode is CONNECTION_GROUP_REQUEST. The method checks to see whether the receiver of the request is a member of the group specified by the sender field of the message. If it is, the request is from the replication group of the replica that received this message, and is either stored in the *MajorityDelayBuffer* (if the replica has not yet generated the identical request), or removed from the *MajorityBuffer* (if the replica has already generated the request, and has stored it in case it needs to recover from a leader crash failure).

On the other hand, if the receiver of the request is a member of the group specified by the receiver field of the message, the request is from the other replication group. In this case, the opcode CONNECTION_GROUP_REQUEST is first changed to REPLICATION_GROUP_REQUEST. The replica then checks to see whether it has already seen the request. A copy of the request would have been stored in the *MulticastDelayBuffer* if the replica had received a copy of the request from the leader of the replication group (see step 3) before it received the message in the receiver connection group. If the *MulticastDelayBuffer* contains the request, it is removed from the buffer. Otherwise, the request is added to the *TotalOrderBuffer*, to permit recovery if the leader fails before it multicasts the message in the sender replication group. Finally, if the replica is the leader, it multicasts the message to the members of the replication group through the **MulticastToReplicationGroup** call.

Step 3: The third communication step begins with **MulticastToReplicationGroup**. When each replica in the receiver replication group receives the message, **ReceiveReplicationGroupMulticast** is called. Since this is step 3, the opcode of the message is REPLICATION_GROUP_REQUEST. First, the sequence number of the received message is checked to make sure that the message has not already been delivered. If it has not, the request is delivered to the application, and *LastDelivered* of the receiver connection group is set to the message's sequence number. The method then checks whether the *TotalOrderBuffer* contains the request. If it does, the request is removed from the buffer, since the message has now been delivered to all replication group members. If it does not, the request is added to the *MulticastDelayBuffer* (so it will not be placed in the *TotalOrderBuffer* when it is later received in the connection group (see **ReceiveConnectionGroupMulticast**)).

Steps 4-6 are similar to steps 1-3, but there are several differences that are worth noting. First, different opcodes are used. Second, in step 4, when a reply is ready, **SendReply** is used to multicast the reply in the replication group. In **SendReply**, there is no need to check for a reply in the *ReplyBuffer*, as was done in **SendRequest**. Third, in step 6, a check must be done to determine whether the request corresponding to the reply has been issued by the receiving replica. If it has, the reply can be delivered to the application. Otherwise, the message is placed in the *ReplyBuffer* to be held until the replica produces the corresponding request.

Using the above communication steps, requests and replies are reliably transmitted between pairs of groups of replicated objects, thus ensuring that remote method invocations are reliable, in spite of application value faults and process crash failures.

4. MAJORITY VOTING SCHEME

The communication scheme presented in the previous section provides a method to transmit requests and replies between replicas reliably. In order to tolerate value faults in applications, the scheme uses voting within each replication group. As seen in the previous section, voting is performed on all IIOP messages before they leave a replication group. The voting algorithms we present allow the membership of a replication group and/or the majority size to change during a vote. To vote correctly when these values can change, the voting scheme must decide which replicas will vote on a particular request/reply, how long to wait for each request/reply, whether or not a majority exists, when to report value-faulty replicas to the dependability manager, and when to update the new majority size. In this section, we present algorithms that address these issues. Before describing the algorithms, we describe the two problems that had to be addressed in designing them.

4.1. The Group Membership Change Problem

The voting mechanism that we have developed tolerates value faults in applications in the presence of crash failures. In order to tolerate m replicas that have value faults and n replicas that crash simultaneously, at least $2m + n + 1$ replicas must be created, and a majority exists if $m + 1$ replicas agree on the same value. Using this scheme, when a replica generates a value that is different from the majority value, it should be considered faulty and be killed, and a new replica should be created and added to the group. Similarly, when a replica crashes, it should be removed from the group, and a new replica should be created and join the group. Replica crash failures, the removal of faulty replicas, and the creation of new replicas all lead to group membership changes. In each case, the voting process is affected, since the number of votes that will be cast depends on the number of replicas in the group.

To illustrate how the voting process may be affected, Figure 7 presents a sequence of membership changes that may occur during voting. Suppose three replicas, R1, R2, and R3, exist (as shown in (a)) and are used to tolerate one value fault. Suppose that R3 becomes faulty and generates incorrect messages. The state of the group is then as shown in (b). Next, suppose that a voter detects that R3 is value-faulty, and instructs the dependability manager to kill R3. R3 is then killed and stops generating messages. Furthermore, in order to recover from the value fault, R4 is created to replace R3 and waits to join the group. However, it takes some time for the knowledge that R3 has been killed, and, in turn, knowledge of the corresponding change in the group size, to be propagated to the other group members (See Subfigure (c)). In (c), the group size is still 3, but only two group members generate messages and are thus involved in the ongoing vote.

At the next view change, suppose R3 is removed from the group as shown in (d). In (e), a new view change occurs when R4 joins the group. In order to guarantee replica data consistency, if a new replica joins a group, it must obtain the state from an existing replica. Thus, R4 cannot generate messages until it receives its initial state from another group member. Before this occurs, as shown in (e), the group size is 3, but only two group members are involved in the ongoing vote. Finally, R4 finishes the state transfer and is ready to process messages. The state of the group is changed from (e) to (f), in which all group members are involved in the ongoing vote. Similarly, when a replica crashes, the state of the group changes according to the steps shown in (a), (c), (d), (e), and (f). The only difference is that R4 is created after R3 is removed from the group, and there is a period of time between the crash itself and the eventual change of the group membership. Note that the group size and the number of replicas participating in voting are not directly related. We can thus see that when the group membership changes pating in voting are not directly related. We can thus see that when the group membership changes dynamically, the group size cannot be used as the number of replicas participating in a vote.

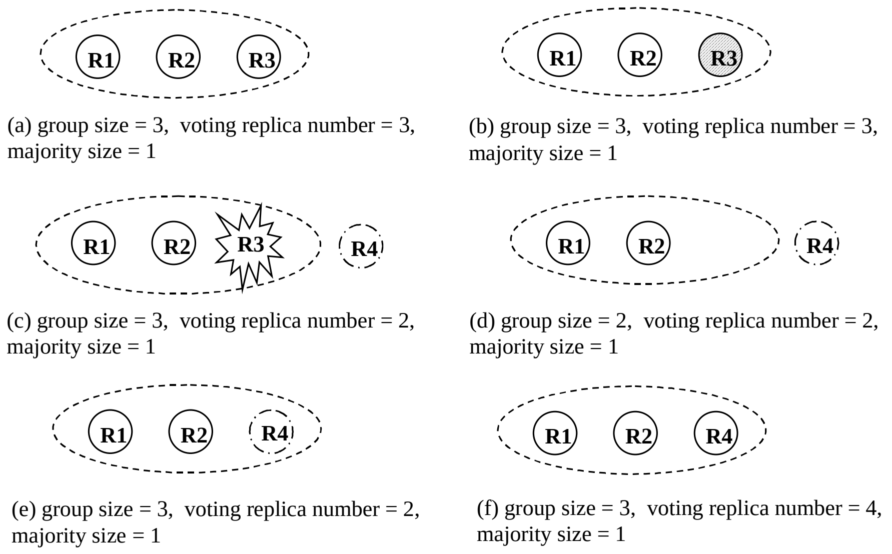

R1
R2
R3
(a) group size = 3, voting replica number = 3, majority size = 1
R1
R2
R3
(b) group size = 3, voting replica number = 3, majority size = 1
R1
R2
R3
R4
(c) group size = 3, voting replica number = 2, majority size = 1
R1
R2
R4
(d) group size = 2, voting replica number = 2, majority size = 1
R1
R2
R4
(e) group size = 3, voting replica number = 2, majority size = 1
R1
R2
R4
(f) group size = 3, voting replica number = 4, majority size = 1

Figure 7. Group Membership Changes due to Crash Failures and Value Faults

Furthermore, the fact that the group size cannot be used as an indication of the number of replicas participating in a particular vote leads to uncertainty on the length of time to wait for each replica's request/reply. For example, in (c), if R3 is just slow instead of dead, a delayed request/reply will eventually be generated by R3, and the number of replicas participating in the vote will be three instead of two. Note that the majority value can be determined before the slow request/reply is received, but delayed requests/replies should be processed even after a majority value has been reached, to detect value faults. For this reason, our scheme processes requests/replies that are received even after a majority value has been reached. The algorithm thus requires a mechanism to handle the delayed messages that contain incorrect values. Furthermore, when no majority value exists among the received requests/replies, the voting scheme must be able to decide either to continue to wait and see if more requests/replies come in, or to stop on the assumption that no more replicas will send their messages. As a result, it is difficult, but also very important, for the voting scheme to determine the number of replicas that will eventually send a request/reply.

4.2. The Majority Size Change Problem

In addition to supporting changes in a group's size during voting, a voting mechanism for a system providing adaptive fault tolerance must support changes in the majority size. A majority size change may occur if applications can dynamically modify the number of value faults to tolerate during execution. For example, when an application increases the number of value faults to tolerate from m_1 to m_2, $2 * (m_2 - m_1)$ new replicas are needed. Figure 8 illustrates the sequence of steps that occur in response to a change in requested dependability when the number of value faults to tolerate is changed from 1 to 2. Suppose that, as shown in (a), R1, R2, and R3 are members of the replication group, and one value fault can be tolerated (the majority size is 2). Assume that an application then increases the number of value faults to tolerate from 1 to 2. Two more replicas (R4 and R5) must thus be created to meet the increased dependability requirement (as shown in (b)). To meet the requirement, R4 and R5 join the group and wait to obtain their states from an existing replica (as shown in (c)). Once R4 and R5 receive state transfers from an existing replica, all five replicas are ready to process requests/replies (as shown in (d)).

As illustrated in (b) to (d), after a request to change the level of dependability is received, time is needed for new replicas to be created, join the group, and become ready to process messages. Since the other group replicas are not blocked during this process and continue to vote on requests/replies, it is important to update the majority size used for each ongoing vote at the appropriate time. For example, if the majority size of the voter is increased from 2 to 3 in (b), (c), or (d) before the new replicas begin to generate messages, not even one value fault can be tolerated. As an illustration, suppose that replicas R4 and R5 join the group and receive their state transfers (as shown in (d)), but receive their states from a replica that has already sent a reply to an ongoing vote. They will thus not participate in ongoing votes. If, however, the majority size for the vote is updated to 3, and one value fault occurs (so now only 2 of the 3 responding replicas agree), no majority will be reached. When this is detected, the voter will generate a false alarm to the dependability manager (stating that no majority could be reached); this may terminate the service for the application. Clearly, this situation needs to be avoided.

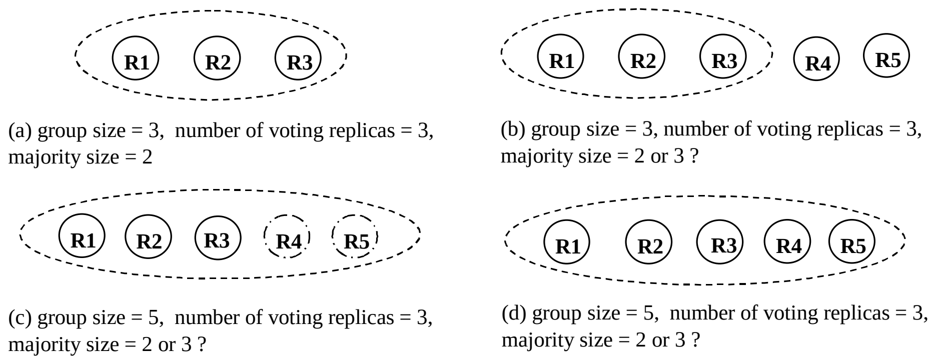

R1
R2
R3
(a) group size = 3, number of voting replicas = 3, majority size = 2
R1
R2
R3
R4
R5
(b) group size = 3, number of voting replicas = 3, majority size = 2 or 3 ?
R1
R2
R3
R4
R5
(c) group size = 5, number of voting replicas = 3, majority size = 2 or 3 ?
R1
R2
R3
R4
R5
(d) group size = 5, number of voting replicas = 3, majority size = 2 or 3 ?

Figure 8. Change in Group and Majority Size in Response to a Change in Requested Dependability

Similarly, when an application decreases the number of value faults to tolerate, the superfluous replicas should be killed to allow efficient use of resources. If the majority size is updated after the extra replicas are killed, there might not be enough replicas participating in voting to allow the original majority size to be reached. Thus the required majority size must be updated carefully. Algorithms to do this correctly, as well as adapt correctly to a dynamically changing group membership, are presented in the next subsection.

4.3. Adaptive Majority Voting Algorithms

We now describe the voting algorithms we have implemented in Proteus, and show how they handle the situations presented in the previous subsections. When tolerating value faults and crash failures in AQuA, voting is done for both requests and replies and is implemented in gateway handlers. The group leader of each replication group is responsible for sending the majority value of each request/reply to the remote object. To ensure that crash failures of the leader are tolerated, each voter is replicated in all replication group members. All replicas maintain information regarding the outcomes of votes, so any replica can become the leader. The majority-voting algorithm described below is invoked each time a request/reply is received. Upon its return, it provides a queue of majority values that are ready to be sent out to the remote objects. Before describing the algorithm in detail, we describe the data structures that are used at each replica to maintain information regarding ongoing votes.

4.3.1. Data Structures for Voting

A table, which is indexed by request/reply sequence number, is kept in each gateway handler for each connection group of which the gateway is a member. (Recall that the connection group and sequence number uniquely identify a request or reply.) Each index in the table points to a *sendValues* structure that holds the votes that have been received for a particular request/reply, as well as the additional information necessary to determine when a majority value has been reached, or a value fault has occurred. More specifically, each *sendValues* entry includes *CurrentMsgNumber*, which counts the number of replicas that have sent a request/reply with that sequence number; *Majority*, which is the majority value reached for the requests/replies with that sequence number; *MajoritySize*, which is the majority size for this request/reply; *ActiveReplicaList*, which holds both the list of active replicas that are ready to generate messages and the requests/replies with that sequence number that are being voted on; and *NewReplicaList*, which is the list of new replicas waiting to obtain their states.

The *ActiveReplicaList* and *NewReplicaList* lists are needed to identify precisely, among the current group members, the replicas that are participating in a particular vote. More specifically, these lists are used to distinguish between new replicas that have recently joined the group, but are not participating in this vote, and replicas that are expected to send a vote for this sequence number. The number of replicas in *ActiveReplicaList* thus indicates the number of replicas involved in the voting. Since both the number of replicas and the states of the replicas might be different for each request/reply, each *sendValues* has its own *ActiveReplicaList* and *NewReplicaList*.

An important side effect of tolerating membership changes during voting is that some requests/replies with larger sequence numbers may reach their majorities before some messages with smaller sequence numbers do. This can happen, for example, when an old replica is slow to generate a request or reply, and a new replica becomes ready to generate requests/replies. If this occurs, the request/reply with the higher sequence number could reach its majority first, because of the new replica involved in the voting.

However, in order to ensure the correct processing of requests/replies by the destination object, the majority value must be sent out in increasing sequence number order, for each connection group. In particular, if requests/replies with greater sequence numbers have already reached their majorities, they must be stored until all requests/replies with smaller sequence numbers have reached their majorities. To permit this, the variable *NextNoMajority* is kept for each connection group to record the smallest sequence number of the messages in that connection group that have not yet reached their majority values. Only majority values whose sequence numbers are less than or equal to *NextNoMajority* can thus be placed into the majority queue or multicast to the remote replication group (recall Figure 5). As a result, it is possible that method **MajorityVotingProcess** (again, see Figure 5) will not return any requests or replies to send, even when a majority value has been reached, or that it will return multiple requests/replies in response to being called with a single vote.

Given this background, we can now describe the algorithms that implement adaptive voting in AQuA. In particular, we describe two methods, **MajorityVotingProcess** and **ResetVoterMajoritySize**, that are used by the value/crash communication scheme described in Section 3 to implement the voting process and to change the voter majority size, respectively. In doing so, we show 1) how a voter processes a new request/reply that is received, 2) how a voter reacts to a change in group membership, and 3) how a voter reacts to a change in desired majority size.

4.3.2. New Request or Reply Received by a Voter

Figure 9 presents the sequence of steps that take place in a voter when a new request/reply is received from a replica by a gateway handler. At a high level, the processing consists of updating the data structures described in the previous section in an appropriate manner, and, if a majority value has been reached on one or more requests/replies, returning these requests/replies to the communication scheme method **ReceiveReplicationGroupMulticast**.

Specifically, lines 1.01-1.02 of the method find the connection group associated with this request/reply and check to see whether the *sendValues* data structure for the request/reply exists. If it does not exist, it is created. Then, if the request/reply is the first one to be received from this replica (i.e., the replica is in the *NewReplicaList* for this *sendValues*), lines 1.06-1.13 update the *NewReplicaList*s and *ActiveReplicaList*s for all *sendValues* structures. Since all replicas process messages in increasing sequence number order, we know that the replica that sent this message will process all requests/replies with equal or higher sequence numbers than this request/reply, but none with lower sequence numbers. We thus move the replica from the *NewReplicaList* to the *ActiveReplicaList* for all ongoing votes with sequence numbers greater than or equal to the sequence number of the received request/reply, and delete the replica from all *NewReplicaList*s with sequence numbers less than the received message's sequence number. Lines 1.14-1.18 adjust the majority size for each ongoing vote, if needed, and will be discussed in Section 4.3.4.

Then, lines 1.19-1.20 place the request/reply into the *sendValues*'s *ActiveReplicaList* and increment the *CurrentMsgNumber* by one. If this request/reply's majority value has not previously been reached, the **CheckMajority** method is called (in line 1.22) to check whether a majority has now been reached. Lines 1.24-1.26 are executed only if the majority value has just been reached for the first time and the sequence number of the returned request/reply is equal to *NextNoMajority*. Lines 1.24-1.26 add the request/reply to the majority queue and set *NextNoMajority* to the sequence number of the message with the lowest sequence number that has not reached a majority value. The method **CheckMoreMessageToCast** is called to find all messages with sequence numbers smaller than *NextNoMajority* in the given connection group. Those requests/replies may then be sent out, preserving the sequence number ordering of requests/replies. To do this, the **CheckMoreMessageToCast** call puts all such requests/replies into the majority queue.

```
1.00 MajorityVotingProcess( message m )
1.01     connectionGroup := voter.lookUp( m )
1.02     sendValues := connectionGroup.lookUp( m.SequenceNumber )
1.03     if ( sendValues = NULL )
1.04         sendValues := CreateSendValues( connectionGroup )
1.05     replicaName := sendValues.removeFromNewReplicaList( m.ID )
1.06     if ( replicaName ≠ NULL )
1.07         sendValues.addToActiveReplicaList( replicaName )
1.08         for each checkSendValues in the connectionGroup
1.09             if ( checkSendValues.removeFromNewReplicaList( replicaName ) ≠ NULL )
1.10                 if ( checkSendValues.SequenceNumber > m.SequenceNumber )
1.11                     checkSendValues.addToActiveReplicaList( replicaName )
1.12                 else if ( checkSendValues.SequenceNumber < m.SequenceNumber )
1.13                     CheckReplicaValues( checkSendValues )
1.14         if (( connectionGroup.NewMajoritySize ≠ 0 ) &&
             ( sendValues.ActiveReplicaList.size ≥ connectionGroup.MinActiveReplicaNumber ))
1.15           for each sendValues in the connectionGroup
1.16               if ( sendValues.SequenceNumber > m.SequenceNumber )
1.17                   sendValues.MajoritySize := connectionGroup.NewMajoritySize
1.18           connectionGroup.NewMajoritySize := 0
1.19   sendValues.insert( m )
1.20   sendValues.CurrentMsgNumber++
1.21   if ( sendValues.Majority = NULL )
1.22       sendValues.Majority := sendValues.checkMajority( sendValues.MajoritySize )
1.23       if (( sendValues.Majority ≠ NULL ) &&
             ( m.SequenceNumber = connectionGroup.NextNoMajority ))
1.24           AddMajorityToMajorityQueue( sendValues.Majority )
1.25           connectionGroup.NextNoMajority := connectionGroup.checkNextNoMajority()
1.26           connectionGroup.checkMoreMessageToCast( majorityQueue )
1.27   CheckReplicaValues( sendValues )
1.28   return majorityQueue
```

Figure 9. **MajorityVotingProcess** Algorithm

Finally, **CheckReplicaValues** (Figure 10) is called for this *sendValues* to see if all the votes that are expected for this sequence number have been received. If they have, the method checks to see whether a majority was reached. If no majority was reached, that fact is reported to the dependability manager. If a majority was reached, all of the returned values are examined to determine whether a value fault occurred, by checking to see if any replicas reported values that differ from the majority value. All replicas that have non-majority values are assumed to be value-faulty, and are reported to the dependability manager.

```
CheckReplicaValues( sendValues )
    if (( sendValues.CurrentMsgNumber = sendValues.ActiveReplicaList.Size ) &&
         ( sendValues.NewReplicaList.Size = 0 ))
        if ( sendValues.Majority ≠ NULL )
            sendValues.checkWrongValue( valueFaultBuffer )
        else
            CreateNoMajorityToReport()
        Delete sendValues
```

Figure 10. **CheckReplicaValues** Algorithm

4.3.3. Action upon Group Membership Change

When the view change occurs, all members in the group are notified of the group membership change via a **ViewChange** callback that is issued by Maestro/Ensemble.

```
ViewChange( view newView )
    CheckViewMember( newView )
    if ( NewLeader() )
        if ( Leader )
```

```
            For each message *m* in the *TotalOrderBuffer*
                **MulticastToReplicationGroup**( *m* )
                While (( *Message* := **RemoveRequestFromMajorityBuffer**()) ≠ NULL )
                    ConnectionGroup *connectionGroup* := **FindConnectionGroup**( *Message* )
                    **MulticastToConnectionGroup**( *connectionGroup*, *Message* )
                    *connectionGroup*.*LastMulticast* := *Message*.*SequenceNumber*
                if ( *ValueFaultBuffer* ≠ NULL )
                    **ReportValueFaultToPCSGroup**( *ValueFaults* )
                if ( *noMajorityToReport* ≠ NULL )
                    **ReportNoMajorityToDM**( *noMajorityToReport* )
    if ( *Leader* )
        **SendToPCSGroup**( *newView* )
```

Figure 11. **View Change** Method

The first step of the **ViewChange** method (Figure 11) is the execution of the **CheckViewMember** method. This method, described below, is used to see if a new replica has joined the group, or if an old replica has failed or been killed. The second step of the method consists of checking whether the view has changed because the (old) leader has left the replication group. To do this, a call to the method **NewLeader** checks whether a new leader exists. If a new leader has been elected, the (new) leader multicasts all messages contained in its *TotalOrderBuffer* to its replication group. This is done because those messages were received by the replication group, but not yet successfully multicast by the (old) leader in the replication group. The leader then finds the associated connection group for each message in the *MajorityBuffer* by applying **FindConnectionGroup**, and multicasts the request/reply to the associated connection group. *LastMulticast* is then updated. If value fault notifications are stored in the *ValueFaultBuffer*, these faults are reported to the dependability manager through the PCS group. If no majority has been reached for the stored sequence numbers, **ReportNoMajorityToDM** reports that fact to the dependability manager. Finally, the leader informs the dependability manager of the changed group membership, using the PCS group structure and the **SendToPCSGroup** call.

The **CheckViewMember** method (Figure 12), which is called from the **ViewChange** method, updates the *NewReplicaList*s and *ActiveReplicaList*s for all *sendValues* entries. The method begins by finding the most recently created *sendValues* structure across all connection groups. It then adds each replica that was not in the old group membership list, but is in the new membership list, to the *NewReplicaList* for the most recent *sendValues* structure. The *ActiveReplicaList*s and *NewReplicaList*s from this structure are copied when new *sendValues* structures are created (see Figure 9, line 1.04). This specifies that the new replicas may provide votes for new requests/replies, but that they will not participate in votes that are already in progress. Then, replicas that were present in the old view, but are no longer present in the new view, must 1) be removed from the *ValueFaultBuffer*, since notifications regarding value faults for these replicas are no longer needed, and 2) be removed from the *ActiveReplicaList*s and *NewRepli-*

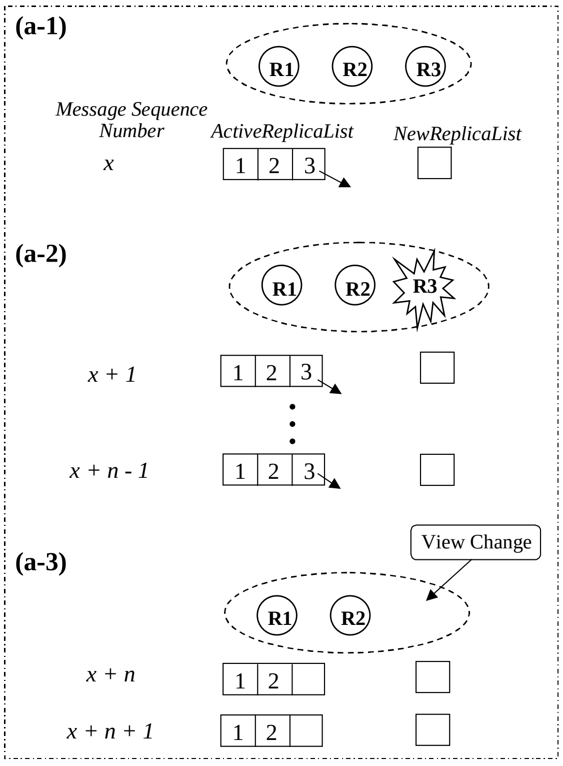
*caList*s. These are replicas that have crashed, or have been killed by the dependability manager. Finally, each *sendValues* structure must be checked, via a call to **CheckReplicaValues**, to see if all requests/replies that will be received for each vote have been received, and, if so, whether a no-majority notification or value fault notification should be sent to the dependability manager.

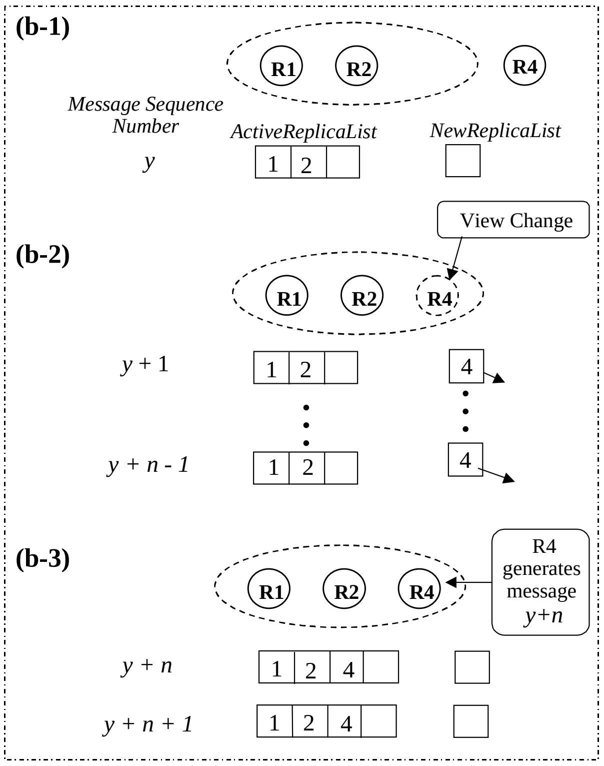
```
CheckViewMember( newView )
    sendValues := LocateTheMostRecentSendValues()
    for each newReplicaName in the newView
        replicaName := RemoveFromOldView( newReplicaName )
        if ( replicaName = NULL )
            sendValues.addToNewReplicaList( newReplicaName )
    for each replica in the oldView
            RemoveFromValueFaultBuffer( replica )
            for each connectionGroup in the voter
                for each sendValues in the connectionGroup
                    sendValues.removeFromActiveReplicaList( replica )
                    sendValues.removeFromNewReplicaList( replica )
                    CheckReplicaValues( sendValues )
    oldView := newView
```

Figure 12. **CheckViewMember** Algorithm

From the above description, we can see that changes in the *ActiveReplicaList*s and *NewReplicaLists* are made to reflect the dynamic group membership changes. By using these lists, the voting scheme can adapt to group membership changes by keeping track of each replica’s state (waiting for state, ready to generate messages, or dead) for each request/reply. Thus, the voting scheme can decide to continue to wait for delayed messages (if more votes are expected to arrive) or stop waiting (if a view change indicates that no more votes will come in). In this way, we can avoid the problems presented in Section 4.1, and correctly vote when membership changes occur during voting.

Figure 13 illustrates how the algorithm just described can tolerate the membership changes illustrated in Figure 7 in Section 4.1. Figure 13 (a) shows a replica leaving the replication group and Figure 13 (b) shows a new replica joining the replication group (*x* and *y* represent sequence numbers of messages). Originally, in Figure 13 (a-1), all three replicas (R1, R2, and R3) generate messages, are involved in the voting, and are thus in the *ActiveReplicaList* (Figure 13 (a-1)). Assume that R3 is killed or fails as shown in Figure 13 (a-2). The remaining replicas keep R3 in their *ActiveReplicaList*s for each vote until they receive the view change callback notifying them that R3 has left the group. At this point, they can safely remove R3 from their *ActiveReplicaList*s for all votes, because no more messages will come from R3 as shown in Figure 13 (a-3). After that, they check whether all messages that will be received for each vote have been received.





(a): R3 Leaves the Group (b): R4 Joins the Group

Figure 13. Effect of Membership Changes in *ActiveReplicaList* and *NewReplicaList*

In Figure 13 (b), a new replica R4 attempts to join the replication group that consists of R1 and R2. As shown in Figure 13 (b-1), R1 and R2 are in the *ActiveReplicaList*, and the *NewReplicaList* is empty. When R4 successfully joins the group, all replicas are notified about the group membership change through a view change callback, and add R4 to their *NewReplicaList*s (Figure 13 (b-2)). The existing replicas will not presume that R4 will participate until they receive a message from R4. Suppose the first message generated by R4 is the message with sequence number *y+n*. When the other replicas receive this message, they will remove R4 from their *NewReplicaList*s with sequence numbers less than *y+n*, and place R4 into their *ActiveReplicaList*s for all ongoing votes with sequence numbers greater than or equal to *y+n* as shown in Figure 13 (b-3). Then, the replicas check whether a majority value has been reached for the ongoing vote with sequence number *y+n*, and whether value faults have occurred for the votes that have received all expected messages. In this way, a vote is expected from R4 in all requests/replies in which it will indeed participate, but not expected from it in any ongoing votes in which it will not participate.

4.3.4. Majority Size Change

Communication from the dependability manager of the new number of value faults to tolerate results in a callback from Ensemble to the **ReceiveReplicationGroupMulticast** method, which was described in Section 3.2. That method then calls the method **ResetVoterMajoritySize** (see Figure 14) after identifying the message type as a *setVoterMajoritySize* message. The routine starts by calculating two values: *newMajoritySize* and *minActiveReplicaNumber*. *newMajoritySize* is the new majority value, which corresponds to the number of value faults that the application requested to be tolerated. *minActiveReplicaNumber* is the minimum number of replicas that is needed to tolerate the number of value faults and crash failures specified by the application. This number is used to ensure that when the majority size is to be increased, it is not increased until enough new replicas join the replication group to support the new majority size.

The method then attempts to set the new majority size for each ongoing vote. Note that the majority size must be set individually for each ongoing vote, to ensure that it will not be changed to a value that cannot be achieved, given the number of replicas participating in the particular vote. In short, the method changes the current majority size only for votes with sequence numbers greater than *NextNoMajority*, and only when there are enough replicas participating in the vote that the new majority size can be reached while the requested number of value faults and crash failures is tolerated (i.e., the number of active replicas for the vote is greater than or equal to *minActiveReplicaNumber).*

More specifically, this is done by iterating through each *sendValues* structure in the appropriate connection group, possibly changing the majority size for *sendValues* entries with sequence numbers greater than *NextNoMajority*. For each *sendValues* structure that meets this criterion, lines 2.06-2.13 are executed if the new majority size is less than the old majority size. Since the dependability manager communicates the changes to the replication group before killing now-superfluous replicas, the *majoritySize* for each *sendValues* structure can be updated to *newMajoritySize* immediately. After the majority size is decreased, votes that previously had not reached their majorities may now have reached their majority values. To determine whether this is the case, the **CheckMajority** method is called. As described previously, it returns a set of requests/replies that have reached their majorities, and, if the sequence number of *sendValues* is equal to *NextNoMajority,* places these requests/replies in the majority queue to be multicast in the connection group.

On the other hand, if the new majority size is higher than the previous majority size, the dependability level requirement is increased, and lines 2.15-2.17 are executed. In that case, if the size of the current *sendValues*'s *ActiveReplicaList* is greater than or equal to the *minActiveReplicaNumber*, there are enough active replicas in the replication group to meet the dependability requirement for this vote. The majority size is thus updated to the new majority size for this *sendValues* entry. Note that once the condition is true for a particular sequence number, it will be true for all sequence numbers greater than that sequence number. (This is because a replica that has participated in a particular vote will participate in all votes with larger sequence numbers.) Thus, the majority value can be updated either for all ongoing votes, or for none, depending on whether the condition in line 2.15 becomes true sometime during the loop that starts on line 2.04.

The flag *resetNewMajoritySize* keeps track of whether the update can be done. If it can, the *NewMajoritySize* for the connection group is set to 0, to indicate that the update is complete. If it cannot be done, the update is deferred until enough replicas join the group and become active to make it possible to achieve the new majority value. The code that checks this condition is in method **MajorityVotingProcess** (Figure 9) in lines 1.14-1.18. The update will occur when enough replicas join the replication group and become active that there are *minActiveReplicaNumber* voting replicas for the sequence number being processed.

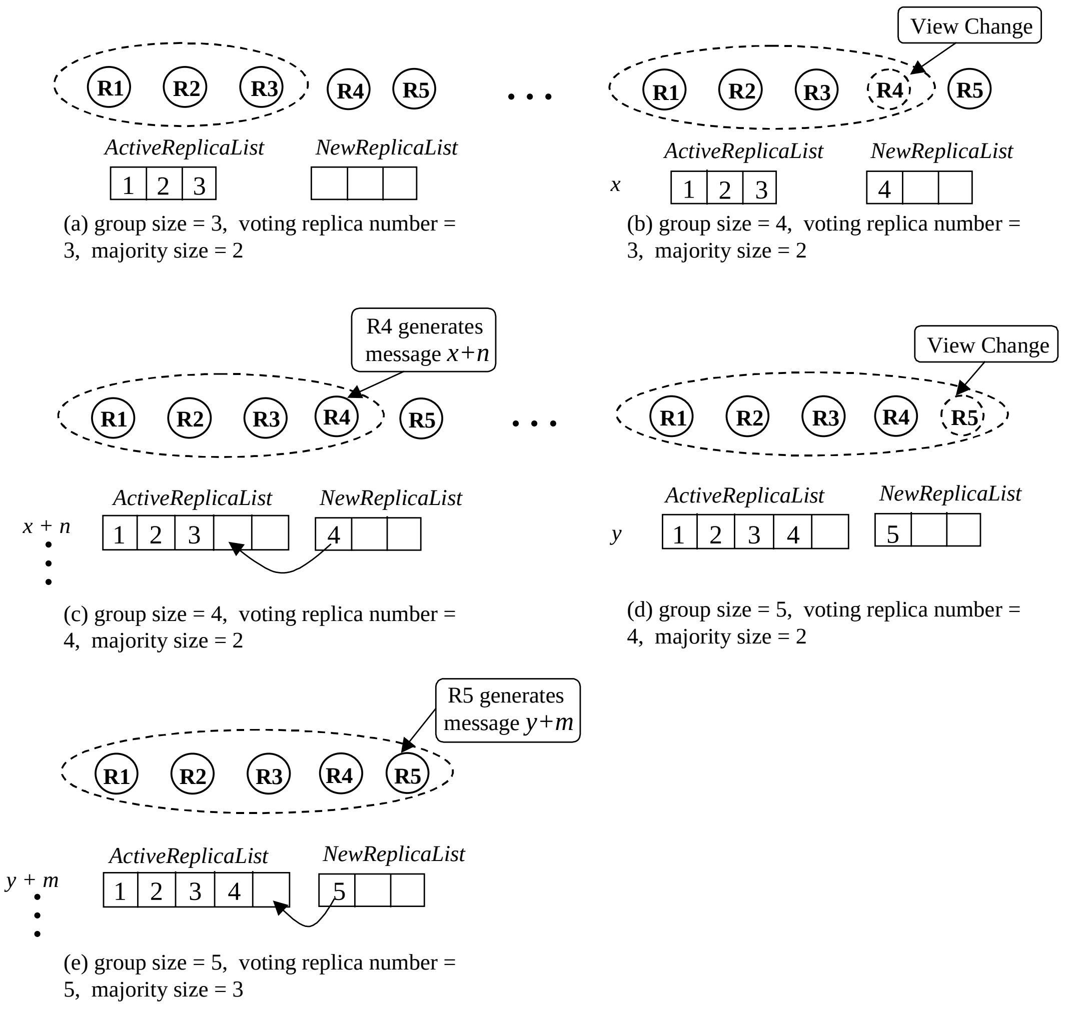
This majority size update algorithm solves the problems raised in Section 4.2 in two ways. First, it specifies the required majority size on a per-vote basis (via a separate value for each *sendValues*'s entry). Second, it only updates the majority size for a particular *sendValues* entry when there are enough active replicas participating in that vote to enable a majority value to be achieved while still tolerating the number of value faults and crash failures specified by the application.

```
2.00 ResetVoterMajoritySize( int valueFaultNumber, int crashFailureNumber )
2.01     connectionGroup.NewMajoritySize := valueFaultNumber + 1
2.02     connectionGroup.MinActiveReplicaNumber := 2 * valueFaultNumber + crashFailureNumber + 1
2.03     flag resetNewMajoritySize := False
2.04     for each sendValues in the connectionGroup
2.05         if ( sendValues.SequenceNumber ≥ connectionGroup.NextNoMajority )
2.06             if ( sendValues.MajoritySize > connectionGroup.NewMajoritySize )
2.07                 resetNewMajoritySize := True
2.08                 sendValues.MajoritySize := connectionGroup.NewMajoritySize
2.09                 sendValues.Majority := sendValues.CheckMajority( sendValues.MajoritySize )
2.10                 if (( sendValues.Majority ≠ NULL ) &&
                       ( sendValues.SequenceNumber = connectionGroup.NextNoMajority ))
2.11                     AddMajorityToMajorityQueue( sendValues.Majority )
2.12                     connectionGroup.nextNoMajorityNumber := connectionGroup.CheckNextNoMajority()
2.13                     connectionGroup.CheckMoreMessageToCast( majorityQueue )
2.14             else if ( sendValues.MajoritySize < connectionGroup.NewMajoritySize )
2.15                 if ( sendValues.ActiveReplicaList.size ≥ connectionGroup.MinActiveReplicaNumber )
2.16                     resetNewMajoritySize := True
2.17                     sendValues.MajoritySize := connectionGroup.NewMajoritySize
2.18     if ( resetNewMajoritySize = True )
2.19         connectionGroup.NewMajoritySize := 0
```

```
2.20        return majorityQueue
```

Figure 14. **ResetVoterMajoritySize** Algorithm

Figure 15 illustrates how to use the above algorithm to tolerate the requested change in dependability given as an example in Section 4.2. In the example, the number of value faults to tolerate is changed from 1 to 2. Figure 15 (a) illustrates the initial state, during which all ongoing votes have lists (as shown), of the *ActiveReplicaList*s and *NewReplicaList*s before the response to this change. Suppose that in response to the request, two more replicas (R4 and R5) are created and that R4 joins the replication group first. As specified by the view change callback algorithm (Figure 11), R4 will then be placed into the *NewReplicaList* of the most recently created *sendValues* structure (Figure 15 (b)). At this point R4 is integrated in the group, but has not yet participated in a vote.



View Change
R1
R2
R3
R4
R5
ActiveReplicaList
NewReplicaList
x
1 2 3
4
(a) group size = 3, voting replica number = 3, majority size = 2
(b) group size = 4, voting replica number = 3, majority size = 2
R4 generates message $x+n$
View Change
$x + n$
1 2 3
4
y
1 2 3 4
5
(c) group size = 4, voting replica number = 4, majority size = 2
(d) group size = 5, voting replica number = 4, majority size = 2
R5 generates message $y+m$
$y + m$
1 2 3 4
5
(e) group size = 5, voting replica number = 5, majority size = 3

Figure 15. Effect of Change in Level of Requested Dependability on *ActiveReplicaList* and *NewReplicaList*

Some time later R4 will participate in a vote by sending a message with a particular sequence number (say *x+n*). At that point, R4 will be removed from the *NewReplicaList* of all *sendValues* structures and be placed in the *ActiveReplicaList* of all *sendValues* structures that have sequence numbers greater than or equal to *x+n* (Figure 15 (c)). This change indicates that R4 should participate in all votes with sequence numbers greater than or equal to *x+n.* R5 follows an identical sequence of steps in joining the group and beginning to participate in votes. (This is shown in Figure 15, parts (d) and (e)). During the transition periods from Figure 15 (a) to Figure 15 (d), the group size is increased, but the majority size must be kept equal to 2 for all the votes until all replicas participating in the votes have received a message from the new replicas. At that point, the majority size can be safely increased from 2 to 3 (Figure 15 (e)).

5. IMPLEMENTATION

We have implemented the AQuA architecture, including the communication and voting algorithms described in the previous sections. The implementation of the gateway (which contains the active replication with majority voting handler) is written in C++, and the dependability manager and object factories are written as standard CORBA applications in Java. The gateway is implemented as a separate process from the application, so that a value fault in the application cannot corrupt the handler. Value fault and crash failure notifications are sent to the dependability manager through internal construction (below the level of the ORB) of CORBA remote method invocations on the dependability manager.

It is relatively easy to make many standard CORBA applications dependable using our implementation of the AQuA architecture. More details on the programmer's interface to AQuA can be found in [Ren99, Bak99]. The type of scheme (type of replication) that should be used depends on the type of faults one wishes to tolerate, the nature of the application (deterministic or non-deterministic), and the fault-free and recovery performance desired. The scheme described in this paper for tolerating value faults and crash failures requires that each replicated object that uses it be deterministic.

Only small changes need to be made to an application object to make it dependable. In particular, when a server application object begins execution, it must make a **registerReference** method invocation on the gateway, passing the gateway its own reference. Similarly, newly created client application objects should use the **getReference** method invocation to obtain the object reference for any remote replicated object with which they wish to communicate. In addition, the object should make a **QoSRequest** method invocation on the dependability manager for each reliable remote object that the new object wishes to use to ensure the creation of a sufficient number of replicas of the particular remote object (if they do not already exist). Finally, an object that makes use of the AQuA infrastructure only needs to implement the **getState** and **setState** methods (i.e., methods that allow the gateway to get and set the state of the object). These methods are used by the active replication with majority voting handler to integrate a new object replica into the group.

To test the validity of our algorithms and their preliminary implementation, we studied several test applications, including the "castle demo" [Ren99]. The castle demo consists of a single undependable client and multiple reliable remote objects (called guards) that interact with the client. Our testbed consisted of five hosts. Up to three guards were running in the castle demo. Several requested levels of dependability were tested: no value fault and up to four crash failures, one value fault and up to two crash failures, and two value faults and no crash failures. Value faults and crash failures were injected in each of these system configurations. In addition, we dynamically switched from one of these configurations to another. Value faults and crash failures were injected in the new system configuration. These tests showed that the algorithm effectively tolerates both value faults and crash failures in an appropriate manner.

In addition, we have used a simple application, called "deet" [Rub00], to benchmark the performance of a preliminary implementation of the developed scheme. The deet application can perform a variety of tests, but in this case was used to make a simple synchronous remote method invocation. The remote method receives an integer as an argument, and returns the integer as a return code. The AQuA code was instrumented to record the round trip time from when the ORB in the gateway passed the IIOP message to the gateway dispatcher until the reply was received by the dispatcher, minus the time needed by the server ORB and application to process the message. In that way, we could determine the overhead in delay caused by the developed replication scheme and group communication subsystem, independent of the processing time taken to execute the remote method itself, and the time spent in the associated ORBs. This measure could thus give us a good indication of the overall overhead added by making the application dependable, and the additional overhead added by the voting replication scheme in particular.

The test setup was as follows. A single (unreliable) client was used to make requests. The server, which implemented the remote method described above, was replicated one to five times, with each copy running on a different host in our testbed. The testbed machines were standard Linux workstations, with processor speeds ranging from 233 to 550 MHZ, connected by a 100 Mb Ethernet. Various dependability requests were made to study the effects of different dependability requirements on fault-free performance. In each case, we ran the instrumented code five times. The average delay times are presented in Table 2. The first row of data in the table presents the baseline case, in which the AQuA infrastructure is present, but the server is not replicated and resides on the same host as the client (1 copy used). Note that in this case the performance of the voting scheme is similar to that of the pass-first scheme (which passes the first reply back to the client object, and hence only tolerates crash failures), requiring approximately 9 ms for round-trip processing across the path described above.

Number of Replicas	Number of Faults to Tolerate	Pass-First Scheme (ms)	Majority-Voting Scheme (ms)
1	0 value faults or 0 crash failures	9.1	9.3
2	1 crash failure	9.7	
3	1 value fault and 0 crash failures		22.7
	2 crash failures	11.6	
4	1 value fault and 1 crash failure		33.1
	3 crash failures	17.9	
5	2 value faults and 0 crash failures		34.4
	4 crash failures	26.5	

Table 2. Experimental Results

The remaining rows of the table present performance results as the number and type of faults to tolerate is changed. As the number of replicas is increased, the performance decreases, since the cost of group communication grows with the size of the group. Furthermore, for a given number of replicas, the voting scheme is slightly more expensive than the pass-first scheme, since multiple replies must be received and voting must occur before a reply can be passed back to the client. In addition, in the pass-first scheme, the requests are sent to the leader directly via a point-to-point message, rather than through a group multicast message as required by the majority-voting scheme. The significance of these overheads depends on the nature of the remote method that is being called. Generally speaking, the overhead caused by the scheme will be minor if the remote method being called is fairly heavyweight, but can be significant if the call itself is lightweight. We are currently reimplementing the AQuA architecture, including the scheme using ACE and the TAO ORB [Sch98], and believe that the new implementation will provide the same functionality with less overhead.

6. RELATED WORK

Other projects have also addressed the problem of making CORBA invocations dependable, and/or providing tolerance to value faults. Several of these efforts are reviewed in this section.

In particular, the Object Management Group (OMG) has recently specified a standard fault-tolerant CORBA architecture [OMG99]. The standard provides fault tolerance based on the replication of CORBA objects. Active replication and passive replication (both warm and cold) are included in the standard. Crash failures are detected through either a "pull-based" or a "push-based" mechanism. In the *pull-based approach*, the mechanism periodically polls applications to check whether the objects are alive, while in the *push-based approach,* the objects periodically send messages to prove that they are still alive. The logging and recovery policies are controlled either by the application or by the infrastructure. Applications are responsible for failure recovery in the *application-controlled approach.* In the *infrastructure-controlled approach,* the standard specifies which mechanism is responsible for logging and recovery. The standard also specifies *fault tolerance domains* that consist of one or more hosts and one or more object groups. The reason for defining these domains is scalability. The standard provides a very flexible way for tolerating CORBA object crash failures. However, the standard does not provide mechanisms to tolerate more complex faults, like value faults.

Several other research projects have also focused on providing fault tolerance to CORBA-based systems. Three main approaches have been developed. In the first approach, a custom ORB is built that includes group communication and fault tolerance mechanisms. Electra [Maf95] chose this “integration” approach. In the second approach, a standard ORB can be used, and group communication and fault tolerance features can be added, through the operation of intercepted IIOP messages. Eternal/Immune [Nar97, Nar99] and AQuA apply this “interception” approach. The third approach involves the addition of group communication and fault tolerance mechanisms as CORBA services. This “service” approach is used by the OpenDREAMS project [Fel96], the Arjuna group [Mor99], and the DOORS project [Gok00].

More specifically, Electra [Maf95] provides fault tolerance to CORBA by building a specialized ORB. The Electra ORB adds several properties of group communication systems to a common ORB. In particular, Electra allows dynamic replication of important object implementations. Moreover, a failure detection service is provided to detect and report failed objects consistently. In addition, Maestro itself [Vay98] provides a CORBA interface. The Maestro approach can be thought of as a combination of the integration and interception approaches. It translates IIOP messages into messages for Ensemble (interception approach), and also provides active replication and manages applications across multiple hosts in the ORB (integration approach).

The Eternal/Immune system adds fault tolerance to CORBA applications by object replication. (Eternal is built on top of Totem [Mos95], while Immune uses Eternal on top of SecureRing [Kih98] and Totem.) Replica consistency is maintained by total ordering of multicast operations, detection of duplicate invocations and responses, transfer of state between replicas, consistent scheduling of concurrent operations, and fulfillment operations for restoring a consistent state after network partitioning and remerging. The Eternal/Immune system contains a “translator” called the Interceptor, which maps between CORBA objects and the group communication (Totem) processes. Immune can tolerate a wide range of faults, including value faults, through the use of SecureRing. For tolerating value faults, Immune votes on client invocations and server responses. The majority size is determined based on the group size, which is stored in a special group (the base group). Immune thus allows voting while dynamic group changes are occurring. However, the majority size cannot be changed dynamically by the application, since it is based on the group membership. Immune provides the user with fault tolerance in a transparent way.

The OpenDREAMS project has focused on the design and implementation of an Object Group Service (OGS), which provides facilities for CORBA object group communication. The mechanisms used to build the group framework are group multicast, dynamic group membership, view change, and state transfer. This approach has the potential to provide group services to CORBA objects; however, it requires that the application developers be aware of, and explicitly make use of, the OGS. Note that [Mor99] uses the same approach, implementing a CORBA object group service by using the Newtop [Ezh95] group communication system. The DOORS [Gok00] project builds a fault-tolerant CORBA service that provides active and passive replication but does not depend on reliable group communication.

Several other projects focus on building dependable distributed systems that can tolerate both value faults and crash failures, but do not use CORBA. Specifically, the Delta-4 project [Pow91, Che92] detects value faults by comparing the signatures of the requests issued by each replica. If some replicas are lost during the voting phase, the phase continues (with the same majority size) if the number of replicas still alive is sufficient for the given majority size. Otherwise, a global error occurs and all replicas are aborted. Delta-4 thus allows voting when the replication group size decreases, but does not seem to provide a framework to allow voting when the majority size changes dynamically. Rampart [Rei95] is a toolkit of protocols for simplifying the development of reliable distributed services in the presence of malicious faults. Rampart contains a set of group communication protocols and output voting protocols. The first output voting protocol is a standard voting protocol, which requires that the client identify and authenticate each server. The second protocol is transparent to clients, but is more costly, because it is based on cryptographic techniques using public and private keys. Finally, the Chameleon project [Kal99] provides a voter ARMOR that votes on an arbitrary number of inputs and returns the majority for an arbitrary K-of-N match. At the voter initialization, a manager specifies the entities from which replies are expected, a timeout period during which these replies can be received, and the entity to which they must be sent. However, Chameleon does not provide a facility for handling dynamic changes of the entities from which results are expected, or dynamic majority size changes.

7. CONCLUSION

We have presented, in the context of the AQuA architecture, an adaptive algorithm to tolerate value faults in applications and crash failures when an application's dependability requirements can change at runtime. The algorithm permits an application to change the level of dependability that it requires, including the number of value faults and crash failures that should be tolerated dynamically during its execution. It also allows the application to adapt to faults that occur in the system. In order to make this possible, the algorithm includes an active replication communication scheme and an adaptive majority voting scheme.

We first described the active replication communication scheme. It consists of six steps for sending each request and receiving the corresponding reply. By using these steps, the communication scheme provides reliable message transmission, totally ordered messages across replicas, and automatic recovery from replica crash failures. We then described two issues that can arise during the majority voting: 1) the fact that the number of replicas can change dynamically during voting, and 2) the fact that majority size changes can occur during voting. We explained how these two issues affect the majority voting process, and proposed an adaptive majority voting scheme to address them. With the adaptive majority voting scheme, voting can proceed correctly while both the group membership and the majority size dynamically change.

REFERENCES

[Bak99] D. E. Bakken, M. E. Berman, M. Cukier, D. A. Karr, J. Megquier, J. Ren, P. Rubel, C. Sabnis, W. H. Sanders, and R. E. Schantz, *AQuA Dependability Management User's and Programmer's Guide Release 2.1*, University of Illinois and BBN Technologies, October 1999.

[Bir96] K. P. Birman, *Building Secure and Reliable Network Applications*, Greenwich, CT: Manning Publications, 1996.

[Che92] M. Chérèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron, "Active Replication in Delta-4," Proc. of the 22nd Annual IEEE Int. Symposium on Fault-Tolerant Computing, pp. 28-37, Boston, MA, July 8-10, 1992.

[Cuk98] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz, "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects," Proc. of the 17th IEEE Symposium on Reliable Distributed Systems, pp. 245-253, West Lafayette, IN, USA, October 1998.

[Fel96] P. Felber, B. Garbinato, and R. Guerraoui, "The Design of a CORBA Group Communication Service," Proc. of the 15th IEEE Symposium on Reliable Distributed Systems, pp. 150-159, Niagara on the Lake, Ontario, Canada, October 1996.

[Gok00] A. Gokhale, B. Natarajan, D. C. Schmidt, and S. Yajnik, "DOORS: Towards High-performance Fault-Tolerant CORBA," Proc. of the 2nd International Symposium on Distributed Objects and Applications (DOA'00), OMG, Antwerp, Belgium, September 2000.

[Hay98] M. G. Hayden, “The Ensemble System,” Ph.D. thesis, Cornell University, 1998.

[Kal99] Z. T. Kalbarczyk, S. Bagchi, K. Whisnant, and R. K. Iyer, “Chameleon: A Software Infrastructure for Adaptive Fault Tolerance,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 6, June 1999, pp. 560-579.

[Kih98] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, “The SecureRing Protocols for Securing Group Communication,” Proc. of the IEEE 31st Annual Hawaii International Conference on System Sciences, vol. 3, pp. 317-326, January 1998.

[Loy98] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken, “Specifying and Measuring Quality of Service in Distributed Object Systems,” Proc. of the First International Symposium on Object-oriented Real-time Distributed Computing (ISORC '98), pp. 43-52, Kyoto, Japan, April 1998.

[Maf95] S. Maffeis, “Run-Time Support for Object-Oriented Distributed Programming,” Ph.D. thesis, University of Zurich, 1995.

[Mor99] G. Morgan, S. K. Shrivastava, P. D. Ezhilchelvan, and M. C. Little, “Design and Implementation of a CORBA Fault-tolerant Object Group Service,” Proc. of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, DAIS’99, Helsinki, June 1999.

[Mos95] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. Lingley-Papadopoulos, and T. P. Archambault, “The Totem System,” *Proc. 25th Annual International Symposium on Fault-Tolerant Computing (FTCS-25)*, pp. 61-66, Pasadena, CA, June 1995.

[Mos98] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan, “Consistent Object Replication in the Eternal System,” *Theory and Practice of Object Systems*, vol. 4, no. 2, 1998, pp. 81-92.

[Nar97] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, “Replica Consistency of CORBA Objects in Partitionable Distributed Systems,” *Distributed Systems Engineering*, vol. 4, no. 3, Sept. 1997, pp. 139-150.

[Nar99] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, “Providing Support for Survivable CORBA Applications with the Immune System,” Proceedings of the IEEE International Conference on Distributed Computing Systems, Austin, TX, May 1999.

[OMG99] Object Management Group, “Fault Tolerant CORBA Specification,” OMG Document orbos/99-12-08 edition, December 1999.

[Pow91] D. Powell, ed., “Delta-4: A Generic Architecture for Dependable Distributed Computing,” *ESPRIT Research Reports*, vol. 1, Springer-Verlag, 1991.

[Pow92] D. Powell, “Failure Mode Assumptions and Assumption Coverage,” Proc. of the 22nd Annual IEEE Int. Symposium on Fault-Tolerant Computing, pp. 386-395, Boston, MA, July 8-10, 1992.

[Rei95] M. K. Reiter, “The Rampart Toolkit for Building High-Integrity Services,” *Theory and Practice in Distributed Systems, Lecture Notes in Computer Science 938*, Springer-Verlag, pp. 99-110, 1995.

[Ren99] J. Ren, M. Cukier, P. Rubel, W. H. Sanders, D. E. Bakken, and D. A. Karr, “Building Dependable Distributed Applications Using AQuA,” Proc. 4th IEEE Symposium on High Assurance Systems Engineering (HASE'99), pp. 189-196, Washington D.C., November 17-19, 1999.

[Rub00] P. G. Rubel, “Passive Replication in the AQuA System,” MS thesis, University of Illinois at Urbana-Champaign, 2000.

[Sab99] C. Sabnis, M. Cukier, J. Ren, P. Rubel, W. H. Sanders, D. E. Bakken, and D. A. Karr, “Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA,” Proc. 7th IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-7), pp. 137-156, San Jose, CA, USA, January 1999.

[Sch98] D. C. Schmidt, D. L. Levine, and S. Mungee, “The design of the TAO real-time object request broker,” Computer Communications, vol.21, no.4, 10 April 1998, pp.294-324. Publisher: Elsevier, UK

[Vay98] A. Vaysburd and K. P. Birman, “The Maestro Approach to Building Reliable Interoperable Distributed Applications with Multiple Execution Styles,” *Theory and Practice of Object Systems,* vol. 4, no. 2, 1998.

[Zin97] J. A. Zinky, D. E. Bakken, and R. E. Schantz, “Architectural Support for Quality of Service for CORBA Objects,” *Theory and Practice of Object Systems*, vol. 3, no. 1, April 1997, pp. 55-73.