Replicated K−Resilient Objects in Arjuna

Mark C Little and Santosh K Shrivastava

Computing Laboratory
University of Newcastle upon Tyne, UK.

Abstract

This paper describes the design of an object replication scheme for the Arjuna distributed system. The design supports K−resiliency, where, in the absence of network partitions, K out of a total of K+1 replica failures can be tolerated before an object becomes unavailable. The scheme chosen employs active replication where each and every functioning replica of an object carries out processing. Computations are structured as atomic actions (atomic transactions). The paper presents the details of how object groups are created and terminated, how a group can be invoked and object replicas inserted and removed in a consistent manner in the presence of node failures.

Key words.

Atomic actions, transactions, active replication, fault−tolerance, object oriented systems, distributed systems, replication.

Appeared in the Proceedings of the First IEEE Workshop on Replicated Data, Houston, November 1990.

1: Introduction

Arjuna is a fault tolerant distributed system designed and built at Newcastle [1, 2]. Arjuna supports nested atomic actions (nested atomic transactions) which are used for controlling operations on objects (instances of C++ classes). Objects are long lived entities (persistent) and are the main repositories for holding system state; they are also the units of replication for increasing availability. This paper describes the design of an object replication scheme for Arjuna. We also examine the problems posed by object replication (as against data replication) and the manner in which they have been dealt with in our design. The design to be presented here supports *K−resiliency*, where in the absence of network partitions, *K* out of *K+1* replica failures can be tolerated before an object becomes unavailable.

2: Replicated Computations

2.1: Object Groups

Data replication techniques for transaction systems to maintain 'one copy serializability' have been studied extensively [eg.3]. Applying these techniques to objects is not as straightforward as would appear at a first glance. The main difficulty arises from the fact that an object is not just data but data (*instance variables*) plus code (*methods or operations* which operate on the instance variables); furthermore, methods can contain calls on other objects. Thus the problem of managing replicated objects really amounts to that of managing *replicated computations*. This problem can be best formulated in terms of the management of object groups (where each group will represent a replicated object) which are interacting via messages. To avoid any consistency problems, it is necessary to ensure that a group appears to behave like a single entity in the presence of concurrent invocations and failures. If not managed properly, concurrent invocations could be serviced in different order by the members of a group, with the consequence that the states of replicas could diverge from each other. Group membership changes (caused by events such as replica failures and insertion of new replicas) can also cause problems if these events are observed in differing order by the users of the group. For example, consider the following scenario (see fig. 1), where object group G_A (replicas A_1, A_2) is invoking an operation on group G_B (a single object B) and B fails during delivery of the reply to G_A. Suppose that the reply message is received by A_1 but not by A_2, in which case the subsequent action taken by A_1 and A_2 can diverge. The problem is caused by the fact that the failure of B has been 'seen' by A_2 and not A_1. To avoid these problems, communication between object groups is typically performed using *ordered reliable multicasts*: reliability ensures that all correctly functioning members of a group receive messages intended for that group and ordering ensures that these messages are received in an identical order at each of the members. ISIS [4], X−kernel/Psync [5] and DELTA−4 [6] are examples of distributed systems relying on such multicasts for group management. In this respect, the group management scheme employed in Arjuna is different: we use unordered reliable multicasts at the communication level and impose ordering at the application level, by relying on the serialization property of atomic actions. Such an approach can have two performance related advantages: (i) it is usually possible to implement faster protocols for reliable unordered delivery as compared to reliable ordered delivery; and (ii) application level ordering through concurrency control enforces order only where necessary (for example, concurrent 'read' invocations on an object group need not be ordered). Thus the design to be described below indicates how object replication can be supported in atomic action based systems without the use of order preserving multicast communication protocols.

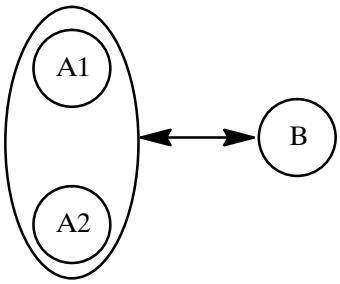

Fig. 1. Object groups

2.2: Failure Assumptions

It is assumed that the hardware components of the system are workstations (nodes), connected by a communication subsystem (for example, a local area network). A node is assumed to possess the *fail−silent* property: it either works as specified or simply stops working (crashes). After a crash a node is repaired within a finite amount of time and made active again. A node may have both stable and non−stable (*volatile*) storage or just non−stable storage. All of the data stored on volatile storage is assumed to be lost when a crash occurs; any data stored on stable storage remains unaffected by a crash. We assume further that there are no network partitions and processes on functioning processors are capable of communicating with each other in bounded time.

The message delivery property between processes can be met realistically if the communication network does not get congested causing messages to be transported extremely slowly and the network interface of each host contains a sufficiently powerful processor with enough memory such that not only every message correctly delivered by the network is acknowledged but also a delivered message is not subsequently lost due to buffer space shortages. This means that a bounded number of retransmissions to get acknowledgements are assumed to be sufficient for a functioning processor to be able to deliver a message to other functioning processors. Such a communication system architecture will be assumed in our discussions whenever message protocols are discussed. If the message delivery property cannot be met realistically then it is possible for a process on a functioning processor to occasionally miss receiving messages directed to it. Such a behaviour can create inconsistencies; for example, process p_j could appear failed to some process p_i and functioning to its replica p_k, in which case the states of replicas p_i and p_k could diverge. The replication scheme presented here cannot be expected to perform correctly under such situations. A solution then is to abandon the quest for K−resiliency using K+1 replicas and opt for *voting or quorum* based replication techniques, which require at least K+1 out of 2K+1 replicas to be functioning in order to tolerate a maximum of K replica failures.

3: The Object Replication Scheme

3.1: Overall Design

The scheme chosen employs *active replication* whereby each and every correctly functioning member of a group performs processing; this is in contrast to the *passive replication* scheme, where only one of the group members, the coordinator, performs processing and checkpoints its state to the passive replicas [eg. 4]. We assume that computations performed by objects are *deterministic* so that if all the functioning replicas of an object have identical initial states then they will continue to produce identical responses to invocations provided the invocations are executed in an identical order.

Every persistent object in Arjuna has a 'home node' where it normally resides in a *passive state* in a (stable) object store; it is made *active* once it comes into the scope of a client

computation. Activating an object entails bringing the state of the object into the home node's primary memory using a primitive operation *initiate* (− −) and linking it to the object's methods; a server process is also associated with the object to serve invocations. Once activated, an object stays that way ready to receive invocations. When a top level commit occurs, the current state of the object is forced back to the stable store. A complementary operation to *initiate*, called *terminate* is available for passivating an activated object (and destroying the server process). The simple example below illustrates the relationships between activation, termination and commitment:

```
{
    O1 objct1           /* objct1 activated
    O2 objct2           /* objct2 activated
    AtomicAction act
    act.begin()         /* start of atomic action act
    objct1.op(...)
    objct2.op(...)      /* invocations ....
    .......
    act.end()           /* act commits
}                   /* object1, objct2 passivated
```

Operation invocations are by means of *remote procedure calls* (RPCs). A naming system maintains the mapping between object names and locations (hostnames). The task of locating a (passive) object and activating it before invocations has been automated through a C++ stub generator which also performs parameter marshalling and other functions necessary for making RPCs [8]. As stated before, atomic actions can be nested; the commitment of an outer most action −*the top level action*− is responsible for making any state changes made to persistent objects stable and releasing the locks on the objects. If desired, an atomic action can invoke (start) a top level atomic action, not nested within the invoking action, which can commit independent of the invoker.

To manage replicated objects, we must provide the following additional functionality: (i) the states of all the passive replicas of an object on functioning nodes must be identical; (ii) object activation should activate all of these replicas; (iii) once activated, the group of replicas should behave like a single entity; (iv) replicas on a failed node are properly initialised during the node's recovery, before they become available again for subsequent use; and (v) it should be possible to dynamically vary the degree of replication of an object (increase or decrease the number of replicas).

The above functionality has been incorporated in our scheme using four closely related mechanisms concerned with group view maintenance, reliable group communications, exclude list and available list management and replica insertion and deletion. These are described here, while some additional design details appear in the next section.

(i) *Group view*: A highly available 'groupview' database is maintained as a part of the naming system. For every replicated object, this database maintains two lists: (1) an *available list* containing the names of the nodes where *potentially available* replicas reside; a potentially available replica is available for use provided its node is functioning (note that, since node failures cannot be detected instantaneously, the available lists held in the groupview database can never be guaranteed to be up−to−date); and (2) a *use list* which contains entries of the form $<N_i,C_i>$, where C_i counts the number of users of the object from node N_i. An empty *use list* for an object indicates that the object is not currently in use. Groupview information is updated− as unavailable replicas are removed from available lists and newly available replicas are inserted in the available lists− using atomic actions to ensure that view changes are 'observed' consistently by other atomic actions. As we shall see, the database is structured as an Arjuna class *groupview*; the database can itself be

replicated to ensure high availability using the techniques to be described here. The class exports several operations:

- *getview (objectname)*: returns to the caller the *available list* for object *objectname*. As a side effect of this call, the *use list* of *objectname* is updated (a new entry is made, with count value equal to one, if there was no entry for *hostname*, or the count value of the existing entry is incremented by one). This operation is invoked during the activation of an object.

- *exclude (<objectname$_1$, nodelist$_1$>, <objectname$_2$, nodelist$_2$>, ...)*, where <objectname$_i$, nodelist$_i$> is the *exclude list* for objectname$_i$: for every *objectname$_i$*, the corresponding available list is modified to remove the entries for the hostnames listed in *nodelist$_i$*.

- *include (objectname, hostname)*: if the *use list* of *objectname* is empty and the *hostname* is already in the available list then return *not−modified*; if the *use list* of *objectname* is empty and the *hostname* is not in the available list for *objectname*, then insert the *hostname* in the available list and return *modified*; if the use list for *objectname* is not empty then then return *trylater*.

- *remove (objectname, hostname)*: if the *use list* for *objectname* is empty then remove the entry (if any) for the *hostname* from the available list, provided the available list does not become empty; else return *trylater*.

- *release (objectname, hostname)*: select the entry for *hostname* from the *use list* kept for *objectname*, and decrement the count by one.

- *recover (hostname)*: remove the entries for the *hostname* from all the *use lists* kept in the database.

All the operations require a write lock as their executions need to be mutually exclusive; their use for maintaining the lists of available objects will be described below. The groupview database is used by every application level atomic action accessing replicated objects, so it is necessary to prevent it from becoming an access bottleneck. Our design strives to achieve this by ensuring that the database is not kept locked for long durations; in particular a lock on it is not held for the duration of the application level atomic action (which would be unacceptable if such actions lasted a long time), but taken and released at the start and end of each action.

(ii) *Reliable group communications*: Reliable multicast communication is employed for (i) implementing *initiate* (...) for activating replicated objects; and (ii) invoking object group through a *multicall (groupname ....)* RPC primitive. *Initiate* tries to activate all the replicas (whose hostnames have been obtained from the groupview database using the operation *getview*), returning the list of hostnames where the activation does succeed. The invoker of the initiate primitive can thus find out the names of hosts containing replicas which are down. All the activated replicas automatically become members of a freshly created group, whose name is supplied as a parameter of *initiate*. Once thus activated, the replicas can be invoked using the group name. The invocation scheme provides a reliable invocation service which ensures that if an invocation made from object group G_A to object group G_B (made using the primitive *multicall* (*groupname ...)*) is received by some functioning member of G_B then all the other functioning members of G_B will also receive the invocation and if a reply sent from G_B to G_A (using a primitive *sendresult (...)*) is received by some functioning member of G_A then all the other functioning members of G_A will receive the reply as well (thus scenarios such as that discussed with respect to fig. 1. where A_1 gets the reply but not A_2 are prevented from occurring). Objects receive invocations through a primitive *getwork (...)* which filters out any duplicate requests coming from a group.

(iii) *exclude and available lists management*: Atomic actions activating and invoking groups can dynamically acquire information on node failures (for example, during *initiate* as mentioned before). For every replicated object an action modifies, it maintains in a list called the *exclude list* the names of nodes (if any) that have been detected to have failed. At (top−level) action commit time, these lists are used for updating the groupview database using the *exclude (...)* operation, for removing the names of the nodes recorded in the exclude lists from all the corresponding available lists.

(iv) *node recovery, inserting and removing object replicas*: To start with, a recovering node calls the *recover* operation of the groupview database to remove any entries kept for this node in any of the use lists. This is necessary as these entries record pre−crash usage information which is now out of date. Further, for all the replicas residing on a node's object store, the node must ensure that the states of these replicas are made identical to those on functioning nodes. One straightforward means of getting the current state of a replicated object is for a recovering node *i* to read the stable state from any of the node *j* listed in the available list provided the object is passive and not in use (ie.the use list is empty). Thus, node i executes the *include* operation for every replica it is maintaining; if the response is *not−modified* then no further action is necessary for that replica, since no modifications have taken place and the replica has not been excluded. If *modified* response is obtained, then the replica is updated by obtaining the state of the object from some node j; if the response is *trylater* then retries are made till either of the previous two responses are obtained and requisite actions taken. A replica thus processed becomes available, that is, the node can accept *initiate* requests for it. A functioning node can remove any of its object replicas by executing the *remove* operation (*remove* ensures that the removal is permitted only if the available list will not become empty). Inclusion and removal of replicas are performed as atomic actions, to prevent any interference with on-going computations.

Consider the execution of an operation of some object A; this operation is to be executed as a top level atomic action; further, the operation contains calls on some object B which is currently passive. Both A and B are replicated. We will describe the (replicated) sequence of activities for object groups for A and B (starting with the activation of B by A). We will assume that object A has been activated, and all its active replicas belong to a group G_A.

- Object B comes in to scope as the execution of the operation of A commences. G_A activates object B using *initiate (...)*, supplying a freshly created group identifier, G_B. As a part of *initiate*, G_A invokes *getview (−B−)* operation of the groupview database to get the available list of (potentially) available nodes containing copies of B. The *getview* operation is invoked as a top−level atomic action (which can commit independent of the invoking action). The available list is used by *initiate* for object activation; G_B gets created. Using the information supplied by *initiate*, the action management system at G_A constructs an *exclude* list of nodes, containing the names of failed hosts (if any) containing replicas of B.

- Operations on B are invoked by G_A making calls to G_B using *multicall (G_B, ...)*; members of G_B receive invocations using *getwork (...)* and send replies using *sendresult (...)*. Assuming there is no client other than G_A, all the members of G_B will service identical invocations in an identical order reaching identical states. Exceptional responses indicating node crashes are used to update the exclude lists held at G_A. The case of multiple groups of clients will be discussed shortly.

- Assume G_A is now committing. In this case G_A invokes, as a part of its commit operation, the *exclude(...)* operation of the groupview database, to remove the names of any failed nodes from the available list of B. Thus the database will now contain the list of replicas of B that are mutually consistent. Assuming that B now goes out of scope, G_A invokes the *terminate* operation to terminate G_B.

We now describe how concurrency control for replicated objects is performed. To start with, objects in Arjuna are responsible for managing their own concurrency (strict two phase locking in the current version [8]); thus the user of an object is not explicitly responsible for obtaining a lock, rather it simply invokes the desired operation of the object and leaves the responsibility of ensuring proper locking to the object. Each operation of an object contains the necessary code for obtaining a read or a write lock. Assume that if a lock cannot be obtained, the operation terminates, returning a response 'locked'. In keeping with the locking policy for nested actions, held locks are released only at the commit time of the top level action. When an object is replicated, the overall effect desired is that of locking the entire group of activated objects. This can be performed by ensuring that an operation invocation terminates at all the replicas in an identical manner resulting in identical responses. Let us consider the previous example, and assume that the operation of B invoked by A requires a write lock; assume further that there is another activated object C (group G_C) that contains a call to B (which also requires a write lock). Thus G_C will also activate object B, so activated replicas of B will belong to two groups, G_B and G_B' (the latter created because of G_C). Operation invocations from G_A and G_C could reach the replicas of B in differing order. There are in fact two possible scenarios to consider:

(i) The invocation from G_A (G_C) succeeds in obtaining (write) locks at all the replicas, as a result the operation execution continues and identical replies are returned from functioning replicas to G_A (G_C); this also means that the invocation from G_C (G_A) terminates with a locked response from all the activated replicas – in which case G_C (G_A) is free to retry that invocation. Since G_A has succeeded in its first invocation to G_B (meaning that G_B has been locked), all subsequent invocations from G_A will give rise to identical responses from replicas – since no lock conflicts will occur (indeed, G_A need not wait for all the replies for an invocation, the first one received being sufficient).

(ii) Both G_A and G_C succeed in locking distinct replicas, with the result that G_A (G_C) will receive 'locked' responses from replicas where locking did not succeed and normal results from replicas where locking did succeed. Clearly, the replicas of B are no longer mutually consistent; the only possible steps at G_A and G_C is to abort the actions causing replicas of B to be restored to original mutually consistent states. An optimization is certainly possible if the number of activated copies of an object is odd: clients which manage to lock only a minority abort, while the client holding a majority of locks retries to obtain the remaining ones (note that a client initiating an object gets to know the membership of the initiated group, so this is always possible).

To summarize: the above concurrency control scheme ensures that 'exclusive write/shared read' policy extends to replicated objects. Once a group gets write locked by a client group, then all subsequent invocations from the client are serviced in identical order at the locked group (since invocations are synchronous RPCs), where identical state changes will occur at the member replicas. On the other hand, if a group is read locked, then the members of the group could receive invocations in different order from concurrent client groups, but this does not cause any problems of state divergence since no state changes are taking place.

From the discussion presented above, we can see that substantial performance improvements can be obtained by distinguishing between two types of termination conditions for a group invocation: (i) *type=all*, which requires replies from all the functioning members of the invoked group; and (ii) *type=one*, where a single reply would suffice. Obviously, this latter type of calls could be executed much faster, taking advantage of faster replicas. There are only three cases where the former type of calls are required. Assume that the client making a call 'knows' whether the called operation is of type *read (update)*, thus requiring a *read (write)* lock. In C++ for example, read only operations of a class can be qualified as *const*, the other operations can then be treated as performing

updates, requiring write locks. Then the three cases requiring *type=all* locks are: (i) having initiated a replicated object, the client is making the very first call, C_1, which will require locking the group; (ii) the client is making a subsequent call C_j, $j>1$, which will require *lock promotion*: this will happen if C_1 required a read lock and C_j is an update operation requiring a write lock, in which case the read lock must be *promoted* to the status of a write lock; a lock promotion must succeed at all the functioning replicas (recall that a read locked object group can be serving more than one client group); and (iii) the client is committing.

3.2: Correctness Properties

The available lists stored in the groupview database satisfy the following *safety property*: the passive states of the object replicas held in the object stores of the nodes whose names are listed in an available list are identical; so it always safe to activate these replicas. As nodes crash, any replicas held in those object stores become unavailable and their states may get stale: such replicas are automatically excluded from the available lists.

The available lists stored in the groupview database also satisfy the following *liveness property*, provided atomic actions using replicated objects eventually commit or abort and every failed node eventually recovers: the hostname of each and every excluded replica eventually gets included in the respective available lists.

The use lists satisfy the following *safety property*: if an object is active (meaning in use), its use list will not be empty. This property is exploited to prevent a replica insertion when an object is in use. The *liveness property* of the use lists states (provided, as before, atomic actions using replicated objects eventually commit or abort and every failed node eventually recovers) that an object not in use will eventually have its use list made empty; thus replica insertion is always possible.

These properties are necessary to ensure that replicas on functioning nodes are in fact available for use, despite occasional node failures. The object replication scheme described in this section somewhat resembles the available copies scheme for data replication [3].

Having presented the overall design of the replication scheme, we will now discuss some specific design details, concentrating on reliable invocation service, exclude list management, node recovery and the construction of replicated groupview database.

4: Design Details

4.1: Reliable Group Communications

A group to group invocation system (say group G_A invoking G_B) can be designed and implemented using 'one to many' multicast communication (see for example the replicated calls in CIRCUS [9]). A client $\alpha \in G_A$ multicasts its call request (C_i) to G_B; this request is executed by all the functioning members of G_B and each such $\beta \in G_B$ multicasts the reply to G_A. Of course, β could get several C_i's (if the G_A has more than one functioning member) which must be filtered out to prevent multiple executions. We require consistent behaviour in the presence of failures such as group member crashes (including failures of entire groups). Just to emphasize this observation, consider the scenario discussed with respect to fig. 1: if G_B fails during a reply, then we want to avoid the situation whereby only A_1 gets the reply and not A_2 (we thus require a stronger consistency property than that provided by the CIRCUS RPC, which cannot cope with this situation). For this reason, we require the reliability guarantee stating that if a call request is received by β then all other functioning

members will also receive it (and vice versa for the corresponding reply message). The primitives referred to earlier, *multicall (...)*, for making a call, and *sendresult (...)*, for sending the reply, provide this functionality by making use of an underlying *reliable multicast communication service*. This service ensures that if a sender's multicast of a message *m* to a group is received by a member of the group, then other functioning members of the group will also receive *m*, even if the sender fails during the multicast. We have designed a multicast protocol, called *rel*/REL_{fifo}, which meets this functionality [10]. Let us first assume the existence of such a service and consider the group invocation protocol (the multicast protocol itself will be discussed briefly afterwards).

The discussion to follow will be in terms of processes – which are the entities which send and receive messages in present day operating systems – so at this stage the reader should perform, where necessary, a one to one mapping from objects to processes. We assume that every process has a message queue where incoming messages, not yet consumed by the process are kept.

(i) *initiate (...)*: Before creating a process group G_B, all functioning $\alpha \in G_A$ must acquire the available list for B (so each functioning α 'knows' on which nodes to create server processes which will belong to G_B). From *initiate* we require the following functionality: for every functioning node named in the available list a server process belonging to G_B gets created and all functioning α acquire identical membership knowledge of G_B. Every functioning α executes the following algorithm: it *individually* invokes the *getview(..B..)* operation as a top level action and obtains the available list (*note*: if there are *n* functioning members in G_A, then *n* such actions will be invoked; this is necessary since entries for each calling node must be made in the use list for B). Let $N = \{N_1 \ldots N_m\}$ be the nodes listed in the available list. For every $N_j \in N$, α sends a 'create server' message (supplying the group identifier G_B) to a well known *manager process* at N_j. If N_j is functioning, a process gets created, which joins G_B (that is, it can receive messages directed at G_B) and takes the necessary steps for activating the local copy of object B; this process then replies to G_A using the reliable multicast. If N_j is not functioning, α is unable to create the server (even after a finite number of retries); eventually, *initiate* terminates, returning the list of names where initiation did succeed. The manager process at each node has the capability of filtering out multiple create requests. Replies from servers are received by every functioning α, which ensures that all members of G_A get identical group view of G_B, since they started with identical available lists. Once G_B has been created, invocations from G_A to G_B are made using reliable multicasts.

(ii) *multicall (G_B, ..., type ...)*: As stated before, there are two termination conditions for a call: *type = all*, meaning get replies from all functioning G_B and *type = one*, meaning a single reply would do. To make the call, α examines its message queue for replies; if the requisite number of replies are not present (ie. from all the members of G_B which are in the clients groupview, for the case when *type = all*, or a single reply for the other case) then α multicasts the call request to G_B. This is repeated a finite number of times to get the appropriate number of replies; *multicall* analyses the replies and sets a return code for the invoker:

OK: normal termination (in particular, for *type=all*, all identical replies received)

failed: no reply received (server group failed)

conflict: different replies received (this response can only be received for a *type=all* call)

The design discussed above can be optimized to reduce the number of message exchanges between G_A and G_B and fully exploit the capability of the reliable multicasting service as follows: suppose that G_A becomes a member of G_B as well (this is trivial to arrange as G_A is

the creator of G_B). Then a call message to G_B sent by α will be received by all other members of G_A; similarly if a reply message is sent to G_B (rather than to G_A) then it will be received by all the servers as well. Thus, a client process sends a call message only if no such message exists in its message queue; similarly a server process sends a reply message (for *type=one* calls) only if its message queue does not contain such a message.

(iii) *terminate (..G_B..)*: use *multicall(..G_B..)* for terminating G_B; then each α *individually* invokes the *release (B..)* operation of the groupview database, as a top level action, to decrement the counter value kept for the calling host in the use list of B .

4.2: Reliable Multicasts

We will now briefly describe the *rel*/*REL*$_{fifo}$ protocol [10] for reliable multicasts (the fifo ordering is for a given sender; a sender's multicasts are received at the destinations in the order they were made). The protocol is simple. Assume the existence of a multicasting service (called *rel*) where by a functioning sender can reliably deliver a message to a group of receivers (one way would be for the sender to individually send the message to the members of the group and receive acknowledgements, and selectively retry this a finite number of times till acknowledgements are received from functioning members; another way, if the underlying network supports broadcasts, like in Ethernet, would be to broadcast the message and follow this up with selective retransmissions till acknowledgements are received). Given the existence of *rel*, the sender uses the procedure *REL* for multicasting a message *m*: *REL* transmits *m* using *rel* twice in a row. So every functioning receiver which receives the second multicast 'knows' that the first one completed successfully, in which case it 'forgets' about this multicast − in the sense that no actions are required for completing the multicast. However, if a receiver, after receiving the first one does not receive the second one within a 'reasonable' timeout period, then it suspects a crash of the sender − in which case there could be a functioning receiver who has not received the message. The receiver, therefore, completes the multicast by (recursively) using *REL* to multicast *m*.

4.3: Constructing and Processing Exclude Lists

Our design requires that atomic actions that update replicated objects ensure that failed replicas get excluded and remain unavailable for subsequent use until such time as specific recovery measures (discussed before) are undertaken for inclusion. The basic steps for replica exclusion were outlined before; here we will elaborate those steps further, highlighting some Arjuna specific design details.

The atomic action subsystem of Arjuna has been constructed in an object−oriented manner, exploiting the type inheritance facilities of C++ [1,2]. Classes for concurrency control, recovery and persistence management (including the atomic action class itself) have been derived from a base class *statemanager*, instances of which act as interfaces to object stores where passive objects are stored. The major advantage of this approach is that specific enhancements to the system can be made by deriving new classes from existing ones. Thus mechanisms needed for constructing exclude lists for replicated objects can be incorporated quite easily. The specific C++ classes necessary for achieving this functionality will not be described here (as it will necessitate understanding the Arjuna class hierarchy which is not strictly necessary for the present discussion on object replication); rather we will describe how the necessary information can be collected and appropriate actions taken.

For every object which is invoked from within an atomic action, the atomic action maintains some state information (eg. hostname and the communication port where invocations can

be sent) which is used for processing a commit or an abort. This information is *merged* with that of the enclosing action when a nested action commits (if however, this action aborts, then the information is discarded). We will now assume that this information contains an exclude list per replicated object which has been accessed from the atomic action. An exclude list for an object contains the names of failed nodes containing replicas of the object. Node failures can be detected whenever: (i) an object is activated using *initiate*; and (ii) an RPC requiring 'all' replies is made. To simplify the discussion on how these lists are processed, we will consider the case of a commit/abort of a top level action containing no nested actions (extension to hierarchic commit/abort being straightforward). Commit processing of some action, say A, involves the following steps (for the sake of simplicity, we are omitting certain optimizations from the description): (i) A sends 'phase one commit' invocation to all the objects accessed; (ii) missing replies from replicated objects are used for updating the exclude lists; (iii) if a 'done' reply is received from every object then the exclude lists of objects that were read locked are deleted and the *exclude* operation is invoked as a top level atomic action, passing all the exclude lists as parameters; after this top level action commits, A proceeds to the phase two of commit processing to terminate the action; (iv) on the other hand, if no reply is received from some object then A decides to abort; this is done in the normal manner, and no exclude operation is invoked.

4.4: Making Groupview Database Highly Available

The replication scheme discussed here requires that the database holding the available lists be available at all times; this requirement can be met realistically if the database itself is maintained in a replicated form. Here we will describe how K−resiliency can be obtained by using a subset of the mechanisms discussed so far.

We will assume that the groupview database is implemented as an Arjuna class *groupview* and n (n>1) instances have been created to get an n−replicated object, *GroupView*. Being an Arjuna object means that *GroupView* can be made persistent and its operations such as *exclude*, *include* can be invoked as atomic actions. All these operations are mutually exclusive and need a write lock. As stated before, where necessary, these operations are invoked by clients as top level actions, so the (replicated) object *GroupView* is kept locked only for short durations (this is important, otherwise *GroupView* could become an access bottleneck). We assume that every node 'knows' the locations of the n copies of *GroupView*. Operations on it are invoked like on any other replicated object, except that no available list is dynamically obtained during *initiate*, rather every invoker tries to perform the operation on all the n copies (whenever *type = all* calls are made) and further no exclude list is prepared. Thus *GroupView* is the only object in the system with a fixed degree of replication (this has to be true for the object which itself is responsible for maintaining the group view information for other objects in the system). We assume the class *groupview* provides one more operation *getstate(...)* which returns the current state. A recovering node N_i containing a copy of *GroupView* invokes this operation as a top level action, which has the effect of locking all the functioning (available) copies before the state of the object is obtained and then updates to the passive state held at N_i's objectstore can be performed.

5: Concluding Remarks

The task of managing replicated objects is in general much harder than that of managing replicated data, because as we have seen, the former task implies managing replicated computations. A general method for supporting active replication is the *state machine* based approach [11], where state machines (objects) use agreement and order protocols for ensuring that the states of correctly functioning replicas do not diverge. The design presented here is much more specific, being applicable to computations structured as

atomic actions running on fail–silent nodes. Principal aspects of our design, namely the reliable group invocation scheme and the scheme for maintaining the group view information were discussed. The group view information itself can be maintained in a replicated object for high availability. The design presented here is currently being implemented. Future work will include evaluating the design by building some applications and experimenting with other forms of replication schemes, such as those based on voting, capable of tolerating a wider class of failures.

Acknowledgements

The work reported here has been supported in part by grants from the UK Science and Engineering Research Council and ESPRIT Project No. 2267 (Integrated Systems Architecture). Extensive discussions with Dan McCue and Xavier Rousset clarified our design.

References

[1] S.K. Shrivastava, G.N. Dixon and G.D. Parrington, *"An overview of Arjuna: a programming system for reliable distributed computing"*, IEEE Software (to appear).

[2] G.N. Dixon, G.D. Parrington, S.K. Shrivastava and S.M. Wheater, *"The treatment of persistent objects in Arjuna"*, The Computer Journal, vol 32, No. 4, pp. 323−332, August 1989.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *"Concurrency Control and Recovery in Database Systems"*, Addison−Wesley, 1987.

[4] K. Birman and T. Joseph, *"Reliable communication in the presence of failures"*, ACM TOCS, 5,1, pp. 47−76, Feb. 1987.

[5] L. Peterson, N.C. Buchholz and R.D. Schlichting, *"Preserving and using context information in interprocess communication"*, ACM TOCS, 7,3, pp. 217−246, Aug. 1989.

[6] D. Powell et. al., *"The Delta−4 approach to dependability in open distributed computing systems"*, Digest of Papers, FTCS−18, Tokyo, pp. 246−251, June 1988.

[7] G.D. Parrington, *"Reliable distributed programming in C++: the Arjuna approach"*, Proc. of USENIX/C++ Conf., San Francisco, pp. 37−50, April 1990.

[8] G.D. Parrington and S.K. Shrivastava, *"Implementing concurrency control for robust Object−oriented systems"*, Proc. of the Second European Conference on Object−Oriented Programming, ECOOP88, pp. 233−249, Oslo, Norway, August 1988.

[9] E. Cooper, *"Replicated distributed programs"*, Proc. of 10th ACM SIGOPS Symposium on Operating System Principles, Washington, pp. 63−78, Dec. 1985.

[10] S.K. Shrivastava and P.D. Ezhilchelvan, *"rel/REL: A family of reliable multicast protocols for high speed networks"*, Tech. Report, Computing Laboratory, University of Newcastle upon Tyne, UK, 1990.

[11] F.B.Schneider, *"The State Machine Approach"*, Section 8.2 in Distributed systems − Methods and Tools for specification: an Advanced course, Lecture Notes in Computer Science, Vol. 190, Springer Verlag, 1985.