Object Replication in Arjuna

Mark C Little and Santosh K Shrivastava
Department of Computing Science
University of Newcastle upon Tyne, UK.

DRAFT: August 1993

Abstract

Arjuna is a fault tolerant distributed system supporting nested atomic actions (nested atomic transactions) that 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 and implementation of two object replication schemes for Arjuna. Support for replication is provided by a naming and binding service for persistent replicated objects that ensures that applications only ever get to use mutually consistent copies of replicas.

Key words

Atomic actions, active replication, passive replication, persistent objects.

1. Introduction

Arjuna is a distributed system, implemented in C++, that provides facilities for constructing applications using persistent objects which can be manipulated under the control of atomic actions (atomic transactions) [1]. Fault tolerance is necessary primarily for preserving the integrity of persistent objects - that is why atomic actions are required. At the application level, objects are the only visible entities; the client and server processes that do the actual work are hidden. Persistent objects not in use are held in a passive state, residing in object stores or object databases and activated on demand. The availability of an object can be increased by replicating it on several nodes and thus storing its state in more than one object store. Object replicas must be managed through appropriate replica-consistency protocols to ensure that the object copies remain mutually consistent.

A replication technique could either be active and passive. In active replication, more than one copy of an object is activated on distinct nodes and all activated copies perform processing [2]. Active replication is often the preferred choice for supporting high availability of responsive services where masking of replica failures with minimum time penalty is considered highly desirable. Since every functioning member performs processing, active replication of an object requires that all the functioning replicas of an object receive identical invocations in an identical order. Thus, active replication requires multicast communication support satisfying rigorous reliability and ordering requirements. Passive replication in its basic form requires that only a single copy (*primary*) be activated; the activated copy regularly checkpoints its state to the object stores where states are stored. One of the advantages of this form of passive replication is that it can be implemented without recourse to any multicast communication (as only one replica carries out the computation at any time); however, its performance in the presence of primary failures can be substantially poorer than under no failures, because the time taken to switch over to a secondary is non-negligible. We therefore believe that a fault tolerant system should be capable of supporting a number of replication schemes.

With this view in mind, we have implemented the necessary infrastructure within Arjuna to enable it to support both active and passive replication of objects. We have done this by implementing a naming and binding service for persistent replicated objects that ensures that applications only ever get to use mutually consistent copies of replicas. The naming and binding service itself has been implemented out of persistent objects whose operations may be invoked under the control of atomic actions; that is how the service is capable of providing the necessary consistency. Using this service, we have implemented the basic scheme for passive replication referred to earlier and a trial implementation of active replication to demonstrate that our system can support both. The passive replication scheme has been made available as a part of the Arjuna system software that has been distributed to other researchers; the scheme has the advantage that it can be supported on top of any 'conventional' RPC system - this is important to a system like Arjuna which has been designed to be capable of exploiting the functionality offered by commonly available distributed computing platforms [3].

2. Background and System Model

2.1. 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). A node may have both stable object store 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 processes on functioning nodes are capable of communicating with each other.

2.2. Objects and atomic actions

A computational model that has been widely advocated for constructing robust distributed systems is based upon the concept of using atomic actions controlling operations on persistent objects. All operation invocations may be controlled by the use of atomic actions which have the well-known properties of (i) serialisability, (ii) failure atomicity, and (iii) permanence of effect. The object and atomic action model provides a natural framework for designing fault-tolerant systems with persistent (long-lived) objects. Persistent objects not in use are normally resident in a *passive* state on object stores. A passive object is made *active* by creating a process (a *server*) for receiving remote procedure calls (RPCs) for method executions and loading the object's state and methods from the object store. Atomic actions are employed to control the state changes to these objects, and the properties of atomic actions given above ensure that only consistent state changes take place, despite shared access and any failures. A number of system services are required to support distributed computations structured out of objects and actions. We enumerate them below:

Atomic Action service: provide atomic action support to application programs.

RPC service: provide an object invocation facility through an RPC mechanism;

Object Storage service: provide a stable storage repository for objects; these object are assigned unique identifiers (UIDs) for naming them;

Naming and binding service: provides a mapping from user-given names of objects to UIDs, and from UIDs to location information such as the identity of the host containing the object store where the state of the object resides.

To be able to access a persistent object, an application program must be able to obtain information about the object's location. The application program can request this information from the naming and binding service by presenting it the name of the object (a string). The naming service can map this string to the UID of the object and then map the UID to the location information. Once the application program (client) has obtained the location of the object it can direct its invocations to that node. It will be the responsibility of that node to activate the object (if the object was in a passive state). Below we discuss this aspect further for the case of replicated objects.

2.3 Object replication

The persistent state of an object typically resides on a single object store of a node; if that node is down, then the object becomes *unavailable*. The *availability* of an object can be increased by replicating it on several nodes and thus storing its state in more than one object store. Such object replicas must then be managed through appropriate replica-consistency protocols to ensure that the object copies remain mutually consistent. We will consider the case of *strong consistency* which requires that all replicas that are regarded as *available* be mutually consistent (so the persistent states of all available replicas are required to be identical). We discuss below three aspects of replica consistency management; the first and the third are concerned mainly with the management of *meta* information about object replicas, whereas the second is concerned mainly with the management of replicas themselves:

(1) *Object binding:* It is necessary to ensure that, when an application program presents the name of an object, which is currently passive, to the naming and binding service, the service returns a list containing information about only those replicas of the object that are (a) mutually consistent, and also (b) contain the latest state of the object. From this information, one, more, or all replicas, depending upon the replication policy in use (see below), can be activated. If the object has been activated already, then given the name of the object, the naming and binding service must permit binding to all of the replicas of the activated object. If we assume a dynamic system permitting changes to the degree of replication for an object (e.g., a new replica for an object can be added to the system), then it is important to ensure that such changes are reflected in the naming and binding service without causing inconsistencies to current users of the object.

(2) *Object activation and access*: A passive object must be activated according to a given replication policy. We identify three basic object replication policies: (i) *Active replication:* In active replication, more than one copy of a passive object is activated on distinct nodes and all activated copies perform processing [2]. So, in the absence of network partitions preventing communication, an object remains available to applications so long as at least one replica is functioning. (ii) *Coordinator-cohort passive replication:* Here, as before, several copies of an object are activated; however only one replica, the *coordinator*, carries out processing [4]. The coordinator regularly checkpoints its state to the remaining replicas, the *cohorts*. If the failure of the coordinator is detected, then the cohorts elect one of them as the new coordinator to continue processing. (iii) *Single copy passive replication*: In contrast to the previous two schemes, only a single copy is activated; the activated copy regularly checkpoints its state to the object stores where states are stored. This checkpointing normally occurs as a part of the commit processing of the application, so if the activated copy fails, then the application must abort the affected atomic action (restarting the action can result in a new copy being activated).

Activated copies of replicas (case (i) and (ii)) must be treated as a single group by the application in a manner which preserves mutual consistency. So, for example, in case of active replication, communication to a replica groups requires reliable distribution and ordering guarantees: 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 functioning member. Of course, it is necessary that the computation performed by each replica is deterministic.

(3) *Commit processing and object passivation*: Once an application has finished using an object, it is necessary to ensure that the new states of mutually consistent object replicas get recorded to their object stores; this takes place during the commit time of the application's atomic action. At the same time, it is also necessary to ensure that the information about object replicas maintained by the naming and binding service remains accurate. Consider an application that modifies some object, say A, and active replication is in use; suppose at the start of the application two replicas for A (A_1 and A_2) are available, but that the crash of a node makes one of them (say A_2) unavailable, so only A_1 gets modified; then at commit time, information maintained about A within the naming and binding service should be modified to 'exclude' A_2 from the list of available replicas of A (else subsequent applications may end up using mutually inconsistent copies of A).

2.4. Client-server bindings

With the above discussion in mind, we present a general model for persistent replicated objects. We assume that for each persistent object there is at least one node (say α) which, if functioning, is capable of running a server for that object (in effect, this would require that the node has access to the executable binary of the code for the object's methods which may well be stored in the object store of α). It is the responsibility of a node, such as α, to bind an incoming object invocation from a client application to the right server. This binding can *break* if the server (client) node crashes; we assume that a broken binding stays that way till the application level action terminates. So in our model, bindings to servers are created only during the start of a client atomic action (as invocations are made); if some bound server subsequently crashes then the corresponding binding is broken and not repaired (even if the server node is functioning again).

Subject to the above binding criterion, a client is bound as follows. If α receives an invocation from the client, and the object is passive at α, then the object needs to be activated before invocation can be performed; this requires creating a server and loading the methods and the state from some object store. We will assume that there is at least one node (say β) whose object store contains the state of the object; we do not require that α be the same as β. So, in order to obtain the state of an object for activation, α may have to contact the naming and binding service to obtain the name of the node (β) storing the object state. We therefore assume that for every persistent object, the naming and binding service maintains two sets of node related data: (i) for an object A, St_A denotes the set of nodes whose object stores contain states of A; and (ii) Sv_A

denotes the set of nodes, each capable of running a server for A. An object can become unavailable if all the nodes $\in Sv_A$ are down (have crashed) and/or all the nodes $\in St_A$ are down. We assume that it is possible to update Sv and St related information for objects maintained by the naming and binding service. For example, it should be possible to exclude a node currently $\in St_A$ if it is found not to contain the latest (committed) state of the object (say the node has crashed); similarly, it should be possible for this node to be included back in St_A once it does contain the latest committed state for the object. We assume that the naming and binding service itself is built out of one or more persistent objects, so the above state transitions are (naturally) performed under the control of atomic actions. The objects providing the naming and binding service themselves can be replicated in order to be able to provide highly available service; we will assume that the service is always available.

We conclude our discussion by illustrating how activation and binding is performed in our model. Consider first the simple case of a non-replicated object (say A) that is currently passive and $|Sv_A| = 1$ and $|St_A| = 1$. Activating A will consist of creating a server at the node $\in Sv_A$ (say α, see fig. 1) and loading the state from the node $\in St_A$ (say β; a common case is $\alpha=\beta$). At commit time, the state of the object is copied back to β. If either α or β are down, or crash during the execution of an atomic action, then the action must abort.

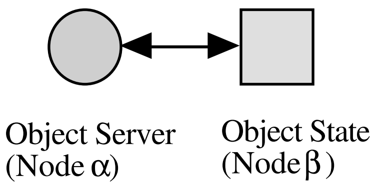

Figure 1: Activating a non-replicated object

The general case is when $|Sv_A| > 1$ and $|St_A| > 1$. Activating A can consist of creating servers at one or more nodes listed in Sv_A (let $Sv_{A'}$, where $Sv_{A'} \subseteq Sv_A$, be the set of such nodes), and loading the state of the object from any node $\in St_A$ (see fig. 2). If $|Sv_{A'}| = 1$, then the scheme will be single copy passive replication; if $|Sv_{A'}| = k$, where $1 < k \leq |Sv_A|$, then the server group of k replicas must be managed for active or coordinator-cohort replication. At commit time, an attempt is made to copy the latest state of the object $\in Sv_{A'}$ to the object stores of all the nodes $\in St_A$. To ensure that St_A contains the names of only those nodes with mutually consistent states of A, the names of all those nodes for which the copy operation failed must be removed from St_A.

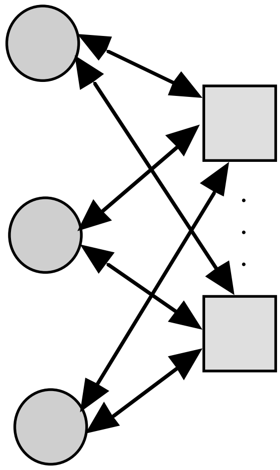

Figure 2: Activating a replicated object

If a client is accessing an object (*A*) which has been activated already, then for the general case of fig. 2, the client must be bound to all the activated copies of *A* at functioning nodes $\in Sv_{A'}$ (or to α, for the case illustrated by fig. 1).

3. An Overview of Object Replication in Arjuna

In the previous section we described three object replication approaches; out of these we have performed implementations of active and single copy passive replication in Arjuna [5, 6]. Our design exploits the fact that concurrency control for atomic actions (strict two-phase locking in Arjuna) will ensure that only objects that are read locked can be shared among different clients. Before describing the main elements of our design, we present some details about Arjuna that will be relevant to our subsequent discussions.

Arjuna provides a set of tools for the construction of fault-tolerant distributed applications constructed according to the model discussed in section 2. In Arjuna a primitive operation *initiate(...)* is used for binding to an object. Arjuna provides an operation to enable a server to load the state of the object from the object store of a node. This operation gets invoked automatically, where necessary, as a side effect of locking the object. Method bodies are expected to contain the relevant (read or write) lock acquisition operations. A complementary operation to *initiate*, called *terminate*, is available for breaking a binding. The Arjuna stub compiler for C++ generates the necessary client-server code for accessing remote objects via RPCs and also generates calls on *initiate* and *terminate* as an object comes and goes out of the scope of a computation [7]. A simplified program fragment is shown below:

```
{
      O1 objct1;         /* bind to objct1 */
      O2 objct2;         /* bind to objct2 */
      AtomicAction act;
      act.begin();       /* start of atomic action act */
      objct1.op(...);
      objct2.op(...);    /* invocations ....*/
      .......
      act.end();         /* act commits */
 }                       /* break bindings to objct1, objct2 */
```

The execution of an Arjuna atomic action thus involves the following sequence of activities: (i) creation of bindings to persistent objects; (ii) start of the atomic action; (iii) method invocations: as a part of a given invocation the object will be locked in read or write mode (assuming no lock conflict), and initialised if necessary with the latest committed state from the object store; (iv) commit/abort of the action; and (v) breaking of the previously created bindings. Variations on this sequence are possible, but for the sake of simplicity, we will only consider programs that give rise to the above sequence. Arjuna ensures that if failures can prevent the completion of an atomic action program, then the atomic action will be aborted, and any 'orphaned' bindings removed.

Again for the sake of simplicity, we shall assume that a client making use of the naming and binding service has already obtained the unique identifier (UID) of the object it is accessing (say UID_A for *A*). The naming and binding service maintains a 'group view database (*GVD*)', that is composed of two distinct persistent objects: an *Object Server* database (the *Name Server*) for maintaining UID_A to Sv_A mappings for all the persistent objects and similarly an *Object State* database for maintaining UID_A to St_A mappings. Clients use the Object Server database, while servers make use of the Object State database. Below we will consider the case of a client accessing some replicated object *A*.

(i) Single copy passive replication: The client contacts the object server database which will either return a list of server nodes where *A* may be activated (*A* is currently passive) or the name of the node where *A* is already active. The client invokes *initiate* for binding, passing the name of any node in the list (A is passive and needs to be activated), or the name of the node where *A* is already active; *initiate* returns a port number for the server managing *A*. For a freshly activated object, the identity of its node is registered in the object server database. The entire set of operations, consisting of the database accesses and binding to a server is carried out within an atomic action. If it is necessary to get the state of the object, the server contacts the object state database to obtain a list containing UID_A to St_A mappings, and loads the state of *A* from any node listed there. From then on, the client action proceeds like an action operating on non-replicated objects. Assume that *A* was updated; then at commit time, the state of *A* is copied on to all the nodes named in the list containing UID_A to St_A mappings; the names of any nodes where this operation failed are removed from the database maintaining UID_A to St_A mappings.

(ii) *Active Replication*: Binding is performed as before, except, *initiate* now returns a group identifier that can be used for sending messages to all the servers managing copies of *A*. The operations *initiate*, *terminate* as well as those for RPC make use of a reliable group communication protocol to ensure message delivery to all functioning members of a server (client) group. A simple group locking protocol is also used for ensuring that all of the members of a group are either read locked or write locked, ensuring 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 (of course, as indicated before, object methods are required to be deterministic). 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. Finally, at commit time, the state server database is updated (if necessary) to exclude any nodes deemed to have failed.

In summary, our approach has been to maintain the basic binding information about object replicas in the GVD which is accessed using atomic actions. However, as we shall see, the GVD is never kept locked for the entire duration of an application level atomic action (for this would be unacceptable, making the database a performance bottleneck). The GVD can be used for supporting either active or passive replication. In the following sections we describe our design in detail.

4. Using the Group View Database

4.1. The Object Server and State Server Databases

Object Server Database: This database maintains two lists per object: (i) for object *A*, its *available server list*, giving the locations (host names) of all the nodes $\in Sv_A$; and (ii) for each node $\in Sv_A$, a *use list*, of the form $<N_i, C_i>$, where C_i counts the number of clients *of* A from node N_i. An object *A* will be in a passive state if, for all nodes $\in Sv_A$, the corresponding *use list* is empty (all $C_i = 0$) .

The operations provided by the database include:

(i) *GetServer(objectname):* for an object, *A*, this operation returns the list of nodes $\in Sv_A$ given its name (UID_A) along with the *use list* (if non empty).

(ii) *Increment(clientnode, objectname, hostname$_1$, ..., hostname$_n$)*: The counter value for *clientnode* within the *use list* of *objectname* for each *hostname$_i$* is incremented by one (if there is no entry for *clientnode*, then a new entry with counter set to one is created).

(iii) *Decrement(clientnode, objectname, hostname$_1$, ..., hostname$_n$)*: A complementary operation to the previous one.

(iv) *Exclude (objectname, hostname$_1$, ..., hostname$_n$):* Remove the *hostnames* from the *available* and *use lists* of *objectname*.

(v) *Insert(objectname, hostname):* This operation is called to add the name of a node that can act as a server site for the named object. This operation succeeds only when the object is passive (*use list* is empty).

(vi) *Remove(objectname, hostname)*: A complementary operation to the previous one.

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

State Server Database: Like the object server database, this database also maintains two lists per object: (i) for object *A*, its *available state list* containing the names of the nodes $\in St_A$; and (ii) for each node $\in St_A$, a *use list* , of the form $<N_i, C_i>$, where C_i counts the number of clients *A* from node N_i. An empty *use list* for an object indicates that the object is not currently in use. The operations of this database are listed below:

(i) *GetState(objectname, hostname)*: returns to the caller the *available state list* for object *objectname*. As a side effect of this call, the *use list* of *objectname* is updated (a new entry is made, with counter value, for each node $\in St_{A'}$ set to one if there was no entry for *hostname*, or for each node $\in St_A$, the counter value of the existing entry is incremented by one).

(ii) *Exclude(objectname, hostname$_1$, ..., hostname$_n$)*:Remove the *hostnames* from the *available* and *use lists* of *objectname*.

(iii) *Release(clientnode, objectname)*: The counter value for *clientnode* within the *use list* of *objectname* is decremented by one.

(iv) *Insert(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 return *trylater*.

(v) *Remove(objectname, hostname)*: if the *use list* for *objectname* is empty then remove the entry (if any) for the *hostname* from the *available list.*

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

4.2. Accessing the GVD

We will now describe how the above database operations are used by clients during the various activities concerned with the execution of an atomic action program (recall that, as discussed at the start of section two, a complete execution of a client program consists of a sequence of five activities). As before, we will consider the case of access to object *A*. The description will refer to fig. 3; we will describe the more general case of active replication (treating passive replication as a special case of active).

(i) Creation of bindings: Within an atomic action (τ_1), the client invokes *GetServer(UID_A)*; the returned information is used for binding using *initiate* as discussed before. If, during *initiate*, any server nodes are suspected to have failed (*initiate* is unable to bind at a given node), then the names of such nodes are removed using the *Exclude* operation of the server database; finally, *Increment* operation of this database is invoked, passing the list of server nodes where the binding did succeed.

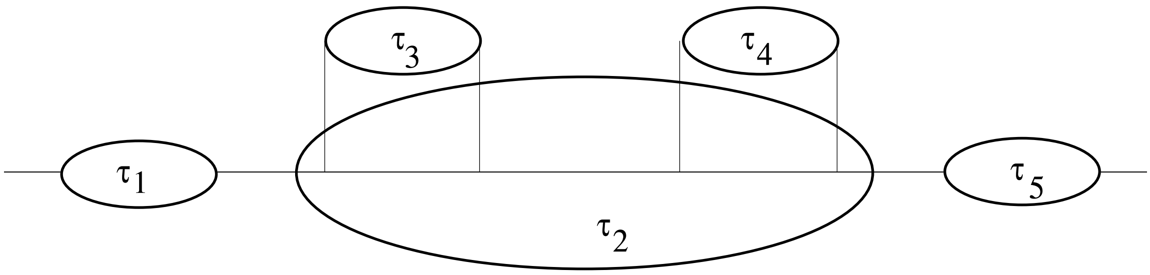

Figure 3: accessing the GVD.

(ii) Loading object state: A server for *A* invokes *GetState(UID_A)* operation to obtain the list of nodes where persistent states for *A* are stored, and then access the object store of any such node to obtain the state of the object. If any node failures are detected, then these nodes are removed by executing the *Exclude* operation of the object state database. All these operations are performed by a server from within an atomic action (τ_3) that is executed as 'nested top level' from within the client action (τ_2). By executing τ_3 as nested top level, it is ensured that locks held on the object state database are released when τ_3 ends (rather than held till the end of τ_2).

(iii) Execution of the action: Any sever failures detected during object invocation are noted by recording their names in an *exclude server list* (such a list need only be kept for the case of active replication only, where an action can commit despite server failures).

(iv) Commit processing: Commit processing involves the following steps : (i) sending 'phase one commit' invocation to all the object servers; (ii) a server returns its yes/no decision, together with an *exclude state list* of nodes where replica states could not be copied to object stores (such a list is not returned by servers that performed read only operations, or are returning a 'no'); (iii) the client launches a nested top level action (τ_4) which operates on the state server database in the following manner: *Release* operation is executed for every object accessed, and in addition, if the client's decision is to commit, *Exclude* operations are executed to remove from the state server database the names of nodes listed in *exclude state lists* ; after τ_4, the client completes the second phase.

(v) Breaking bindings: The client breaks previously created bindings by executing the Decrement operation of the object server database (action $\tau 5$), at this time, the *exclude server list* is used to update the list of available servers.

Other schemes are possible for managing GVD information, and are discussed in [8]. In the trial implementation of active replication in Arjuna, actions such as $\tau 3$, $\tau 4$ were actually executed as nested actions. Passive replication as presently implemented also uses a limited version of the scheme depicted in fig. 3 (since the ability to launch a nested top level action during a commit phase has not yet been implemented): if at commit time the *state exclude list* is found to be non-empty, then the action aborts. If failures are infrequent, then this is not a serious limitation; all this means is that after $\tau 3$, any failures of nodes holding states that would have needed updates at commit cause $\tau 2$ to abort.

4.3. Node Recovery, Inserting and Removing Object Replicas

A recovering node calls the *Recover* operations of the GVD 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. The two databases also maintain a list each of nodes that have been excluded. A functioning node occasionally checks if it has been excluded in error, and if so, performs recovery.

To be able to re-export a possibly excluded server, the recovering node invokes the *Insert* operation on the object server database. Until this operation returns successfully, the given server cannot be used. For all the replica states residing on the node's object store, the recovering node must ensure that these states are brought up-to-date if necessary. So, node *i* executes the *Insert* operation of the state server database 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 previously been excluded. If the *modified* response is obtained, then the replica state 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 functioning node can withdraw its server or state storage facility by executing the relevant *Remove* operation. Inclusion and removal of replicas are performed as atomic actions, to prevent any interference with on-going computations.

4.4. Correctness Properties

The available lists stored in the state server database satisfy the following *safety property*: the committed states of the object replicas held in the object stores of the nodes whose names are listed in an available list are identical and most recent; so it is always safe to use these replicas. As nodes crash, any replica states 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 state server database also satisfy the following *liveness property*, provided atomic actions using replicated objects eventually commit or abort and every failed node eventually recovers, and the database is always available: 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, and the database is available) that an object not in use will eventually have its use list made empty; thus replica insertion is always possible.

Similar properties can be stated for the object server database. These properties are necessary to ensure that replicated objects on functioning nodes are in fact available for use, despite occasional node failures.

5. Managing Active Replicas

In this section we will describe how active replica groups get invoked and concurrency controlled. For the sake of illustration, we will 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.

5.1. Group Invocations

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 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 G_A has more than one functioning member) which must be filtered out to prevent multiple executions. We require the following behaviour despite the presence of failures such as group member crashes (including the failure of an entire group): if a call request is received by some $\beta \in G_B$ then all other functioning members will also receive it (and vice versa for the corresponding reply message). A group invocation scheme with such a property can be implemented by making use of a reliable multicast service which 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. Our group invocation scheme has been implemented using a multicast protocol, called *rel/REL*$_{atomic}$, which provides such a service[10], although any protocol providing similar functionality could be used. The primitives *initiate* and *terminate* also make use of this protocol.

The invocation scheme provides 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 two cases where the former type of calls are required: (i) having initiated a replicated object, the client is making the very first call, c_1, which will require locking all the members of the group, and therefore response from all of them required; (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).

5.2. Group Concurrency Control

The entire group must be locked in a read or write mode. 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). 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 'lock conflict' response from all the 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 not cause lock conflicts (indeed, G_A need not wait for all the replies for a subsequent 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 'lock conflict' 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 optimisation is certainly possible if the number of active 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.

To summarise: 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.

5.3. Making Group View Database Available

The replication schemes discussed here require that the GVD be available at all times; this requirement can be met realistically by replicating GVD as well. To eliminate the requirement of GVD itself requiring a binding system for replicated objects, two techniques have been used: (i) every client is expected to 'know' the locations of GVD replicas; and (ii) client updates to GVD entries are allowed only if a majority of GVD replicas can be updated (else the client action aborts). Thus, a client has to contact a majority to be sure to obtain the most recent information from the GVD.

6. Performance

We have done some simple tests for evaluating the performance of the two replication techniques. A client interacts with an object whose state consists of an integer variable. The class interface is:

```
class ActInt
{
public:
      ActInt ();
      ~ActInt ();

      int Set (int v);
      int Get (int& v);

private:
      int Element;
};
```

The value of `Element` is either modified (`Set`) or inspected (`Get`).

6.1. No Replication

We consider the case where the object is not replicated, is currently passive and the object store storing the state is local to the server node. To give an indication of the overhead involved in using the naming and binding service, we took two timings: one in which the necessary information is 'hardwired' into the client code (so the naming and binding service is not used), and one in which the client must make use of the naming and binding service:

Naming and Binding Service	Time taken to bind to server
Used	0.42 seconds
Not used	0.47 seconds

Table 1: Binding timings

As can be seen from the timings, the overhead involved in contacting the naming and binding service is not appreciable.

We now examine the time taken to interact with the ActInt object once it has been activated, both with and without a server failure (arranged to occur just before commit phase). The timings are for the update operation (the `Set` method).

Server Status	Time taken to commit/abort
Operational	0.36 seconds
Failed	5.40 seconds

Table 2: No replication

To place these figures in context, it should be noted that the time taken for a round-trip null RPC in Arjuna is about 5ms with no failures. The RPC uses a timeout of 5 seconds, and if no response is received, then makes use of the Unix *ping* facility to detect node availability; if no failure is suspected, the RPC continues to wait for another timeout period. Arjuna uses the Unix file system for storing object states and methods. The time taken for a local atomic action operating on an instance of ActInt to commit is about 0.25 seconds. Therefore, it can be seen that the majority of the overhead involved in atomic actions with no failures comes from writing state to the object store. The second timing of 5.40 seconds is a result of the fact that the failure of the object is detected during the commit phase of the atomic action.

6.2. Active Replication

When making use of an active replica group the group had two ActInt members, only one of which was made to fail (just before commit). Each server had access to local object store.

Server Status	Time taken to commit
no failures	0.54 seconds
One replica failed	5.70 seconds

Table 3: Active replication

6.3. Passive Replication

We consider two nodes with object stores, and each such node can run a server. Committing under no failures involves updating both the stores. The figure of 6.71 secs. is obtained when the server crash causes the action to abort; the client then retries, initiates a server at the second node and commits. As expected, under server failures, active replication performs better than passive. Finally, the third figure shows the time taken to commit and exclude a failed node holding object state.

Server Status	Time taken to Commit
no failures	0.53 seconds
Failed primary server	6.71 seconds
Single failed state instance	5.42 seconds

Table 4: Passive replication

Acknowledgements

This work has been supported in part by grants from the UK MOD and the Science and Engineering Research Council (Grant no. GR/H1078) and ESPRIT basic research project 6360 (BROADCAST).

References

[1] S.K. Shrivastava, G. N. Dixon and G.D. Parrington, "An overview of the Arjuna distributed programming system", IEEE Software, January 1991, pp. 66-73.

[2] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", ACM Computing Surveys, 22(4), December 1990, pp. 299-319.

[3] S.K. Shrivastava and D.L. McCue, "Structuring fault-tolerant object systems for modularity in a distributed environment", IEEE Trans. on Parallel and Distributed Systems (to appear).

[4] K.Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems", Proc. of 11th ACM Symposium on Operating System Principles, Austin, November 1987, pp. 123-138.

[5] M.C. Little and S.K. Shrivastava, "Replicated K-Resilient Objects in Arjuna", in Proc. of 1st IEEE Workshop on the Management of Replicated Data, Houston, November 1990, pp. 53-58.

[6] M.C. Little, "Object Replication in a Distributed System", PhD Thesis, University of Newcastle upon Tyne, September 1991.

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

[8] M.C. Little, D. McCue and S.K. Shrivastava, "Maintaining information about persistent replicated objects in a distributed system", ICDCS-13, Pittsburgh, pp. 491-498, May 1993.

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

[10] P.D. Ezhilchelvan and S.K. Shrivastava, "rel/REL: A family of reliable multicast protocols for distributed real-time systems", Tech. Report, Computing Laboratory, University of Newcastle upon Tyne, 1993.