Using Application Specific Knowledge for Configuring Object Replicas

Mark C Little and Santosh K Shrivastava

Department of Computing Science,
University of Newcastle,
Newcastle upon Tyne, NE1 7RU, UK

Appeared in the Proceedings of the IEEE Third International Conference on Configurable Distributed Systems, May 1996

Abstract

In distributed systems, object replication is used to improve the availability and performance of applications in the presence of failures. When determining the configuration of a replicated object (i.e., number and location of replicas), a number of often conflicting factors need to be considered, e.g., the availability and performance requirements of the object. However, application specific knowledge about the objects, such as any inter-dependencies, is typically not accounted for. In many applications this information can affect an object's availability. Distributed systems which support replication typically give users only basic control over the configuration of a replicated object, such as the number or location of individual replicas. Expressing application specific knowledge is not possible. We have designed and implemented a replication sub-system allowing applications to control these aspects of replication. This system allows the efficient replication of an arbitrary number of objects with arbitrary inter-dependencies.

Keywords: replication, dynamic reconfiguration, atomic actions, object-oriented, object dependencies.

1. Introduction

We consider a distributed system in which application programs manipulate persistent (long-lived) objects under the control of atomic actions (atomic transactions). Atomic actions have the well known properties of (i) serialisability, (ii) failure atomicity, and (iii) permanence of effect, and can be nested. Each object is an instance of some class which defines the set of *instance variables* each object will contain and the *methods* that determine the externally visible behaviour. The methods have access to the instance variables and can modify the internal state of the object. It is assumed that, in the absence of concurrent access and failures, the invocation of an object's operation produces consistent (class specific) state changes to the object. Atomic actions then ensure that only consistent state changes to objects take place despite concurrent access and failures. The object and atomic action model provides a natural framework for designing fault-tolerant systems with persistent objects. In this model, a persistent object not in use is assumed to be held in a *passive* state with its state residing in an object store and *activated* on demand (i.e., when an invocation is made) by loading its state and methods from the object store to the volatile store, and associating a server process for receiving RPC invocations. Further, an object is a convenient unit for concurrency control, storage, replication and migration. Arjuna [1], Argus [2], and Guide [3] are just some systems designed broadly according to this model.

Given this model, replication can be used to improve the availability of objects in the presence of failures. Atomic actions guarantee consistency, while replication allows forward progress for an application. A replication protocol is used to guarantee consistency between replicas in the presence of concurrent access and failures [4][5]. With a suitable protocol it is also possible to improve the performance of certain operations, e.g., by reading from a physically close replica.[1]

For the purposes of this paper we shall use the term *replica configuration* to mean the number and location of replicas of an object. There are several issues which arise when determining the number and location of replicas and there is often a trade-off between availability and performance [6][7]. For example, information about the failure characteristics of the components in the distributed system and the read/write ratio of interactions with the object, can be important in the determination of the replication configuration. Application specific knowledge about object inter-dependencies is also important. For example, if object *A* depends upon object *B* for some of its operations, then *A* may only be as available as *B* where these operations are concerned. In many applications it is common for objects to be made up of (or contain references to) other objects, e.g., a *theatre* object that contains references to *seat* objects; a *book* object that contains references to *chapter* objects. An extreme example of this is the object store itself, which is a persistent object that stores other persistent objects. We will refer to objects such as theatre, book and object store as *composite objects* and objects such as *seat* and *chapter* as *constituent objects* (naturally, a constituent object can be a composite object). There is an

1 An un-replicated object will be considered a degenerate case of replication.

implicit dependency between composite and constituent objects, which may need to be considered when determining their replication configuration. In addition, because of application requirements, this configuration information may need to be further augmented. For example, certain seats may be considered to be more important than others, e.g., an executive box. These executive box seats may possess more replicas, or be located on more reliable nodes.

What is required is a means of specifying these object inter-dependencies and other application requirements, which can then be used to determine the optimum replication configuration. Changes in this information, for example as a result of adding another object, may require reconfigurations. We have designed and implemented a replication sub-system which gives applications control over these aspects of replication, allowing applications to control and modify the way in which objects are replicated. This sub-system allows the efficient replication of an arbitrary number of objects with arbitrary inter-dependencies [8][9]. The sub-system has been implemented as part of the *Arjuna* distributed system. To the best of our knowledge, no other system which supports object replication provides similar mechanisms.

1.1 Replication overview

We will consider the case of *strong consistency* which requires that all replicas that are regarded as *available* be mutually consistent. When an object is replicated, the replicas are treated as a single abstract entity, the *replica group* [10][11]. Information about the members of a group is maintained within an object, the *replica group view*, which itself is maintained by a naming and binding service, and referenced by the name of the group (e.g., the application name or system identifier). There are basically three different replication categories:

(i) *active replication*: more than one copy of an object is activated on distinct nodes and all activated copies perform processing [12][13]. Active replication is often the preferred choice for supporting high availability of services where masking of replica failures with minimum time penalty is considered highly desirable. Since every functioning replica performs processing, all object replicas receive identical invocations in an identical order. This is guaranteed by the use of an appropriate ordered reliable message delivery protocol [14]. Of course, it is necessary that the computation performed by each replica is *deterministic*.

(ii) c*oordinator-cohort passive replication:* several copies of an object are activated; however only one replica, the *coordinator*, carries out processing [15]. 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) s*ingle copy passive replication*: only a single copy (the *primary*) is activated; the primary regularly checkpoints its state to the object stores where states are stored [16]. This checkpointing normally occurs as a part of the commit processing of the application; if the primary fails, the affected atomic action must abort (restarting the action can result in a new primary being activated).

2. System model

We will first present a simple model for creating and accessing replicated persistent objects in a distributed environment, and then show how it is possible to enhance it with application specific knowledge of object dependencies and requirements.

2.1 Basic model

We assume that applications use persistent objects under the control of atomic actions. Fault tolerance is necessary primarily for preserving the integrity of persistent objects. 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 their passive states, residing in object stores or object databases and activated on demand. We assume that each persistent object possesses a unique, system given identifier (*Uid*). The disk representation of an object in the object store may differ from its volatile store representation (e.g., pointers may be represented as offsets or object identifiers). Our model assumes that an object is responsible for providing the relevant state transformation operations that enable its state to be stored and retrieved from the object store. The server of an activated object can then use these operations during abort or commit processing to restore/save the state respectively. Further, we assume that each object is responsible for performing appropriate concurrency control to ensure serialisability of atomic actions. In effect this means that each object will have a concurrency control object associated with it. In the case of locking, each method of an object will have an operation for acquiring, if necessary, a (read or write) lock from the associated 'lock manager' object before accessing the object's state; the locks are released when the action commit/abort operations are executed, and the object is deactivated.

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 by contacting a globally accessible (persistent) *naming service* and presenting it the application level name of the object (e.g., 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 is the responsibility of that node to activate the object if it was in a passive state.

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. It is the responsibility of a node, such as α, to bind an incoming object invocation from a client application to the right server. 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 the 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; it is not necessary for α to be the same as β. In order to obtain the state of an object for activation, α may have to contact the naming service to obtain the name of the node storing the object state. Therefore, for every persistent object, the naming service maintains two sets of (persistent) data:

(i) Sv_A: for an object *A*, this set contains the names of nodes each capable of running a server for *A* (the *server group view information*); and,

(ii) St_A: this set contains the names of nodes whose object stores contain states of *A* (the *state group view information*).

This information is the replica group view described in Section 1.1, and is maintained as separate (independent) objects within the naming service, allowing concurrent updates of different group views. There is no assumption about the relationship between Sv_A and St_A information, i.e., it need not be a one-to-one correspondence. 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.

It is possible and necessary to update this Sv and St related information. For example, it is necessary to exclude the name of a node currently $\in St_A$ if the node is found not to contain the latest (committed) state of the object (say the node has crashed); similarly, it is necessary for the name of this node to be included back in St_A once it does contain the latest committed state for the object.

2.1.1 Replica activation

There are several possible options for activating object replicas and creating server and state groups. We shall now present the two ends of this spectrum of activation policies; the interested reader is referred to [17] for a more complete discussion:

(i) $|Sv_A| = |St_A| = 1$: This represents the case of a non-replicated object. Activating *A* will consist of creating a server at the node $\in Sv_A$ (say α) 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 crash during the execution of an atomic action, then the action must abort.

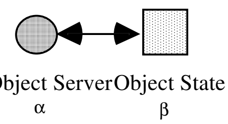

Figure 1: No replication

(ii) $|Sv_A| > 1$ & $|St_A| > 1$: This is the most general case, providing maximum flexibility during object activation. Activating *A* can consist of creating servers at one or more nodes listed in Sv_A. Each server is free to load the object's state from any of the nodes $\in St_A$. At commit time, an attempt is made to copy the state of the object to the object stores of all nodes $\in St_A$. To ensure 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.

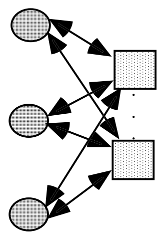

Figure 2: Server and state replication

2.2 Enhanced model

Distributed systems which support replication typically maintain one replica group view for each replicated object, in a manner similar to the model presented above [14][18][19][20]. If each group view is represented as a separate (persistent) object, the overhead imposed in searching the naming service for a specific group view increases with the number of replicated objects[2]; before a group view can be used it must be activated, which requires fetching its state from the persistent store. As a result, systems based upon similar models to that which we have outlined do not scale.

As we have shown in [7], determining object inter-dependencies when replicating objects is also important when attempting to ensure that the (replicated) objects provide a required quality of service. However, the group view information as previously described does not reflect any object inter-dependencies; each group view is used in isolation, and simultaneous reconfigurations to multiple views, for example to modify a composite object and its constituent objects, must be performed at the application level. Typically the application becomes responsible for maintaining information about object inter-dependencies and is also responsible for controlling access to it. Allowing this information to be used in the configuration of a replicated object becomes difficult, because the replication sub-system may have no way of accessing it. Therefore, we believe that this information should be maintained as an integral part of the replication sub-system, and we propose two new types of replica group views (for simplicity in the rest of this paper we shall consider enhancements to St_A information, but similar extensions to Sv_A are possible):

(i) *clustered replica groups*: here a collection of replica group views are clustered with each other within the naming service. Whenever information about a single member of this cluster is obtained from the naming service, information about all clustered members is also obtained and cached locally (similar to the work described in [21]).

(ii) *template replica groups*: information which is common to a series of replicated objects can be factored into a template and only the information which is unique to a group need be stored separately. When replica group information is required from the naming service, the template is *applied* to the group specific information to obtain the final group view. Thus, the amount of information stored by the naming service can be minimised, and if the template is applied to the group view by the client, rather than the naming service, the amount of network traffic can also be reduced).

The replication sub-system should support arbitrary combinations of these group view types to better reflect application and object requirements. In addition, a special template type is also supported: a *wildcard template*. If a group view for an object is requested from the naming service and one does not exist, then the information in this template is used to build a default group view. Using this mechanism it is possible to replicate every object in the system (those that currently exist and those that will ever exist) without having to register them at the naming service.

It is important to understand that only the way in which information necessary to describe and maintain replica groups is changed by the introduction of these new group views; the replica groups are not affected. At the application level, the implementation of a replica group view is transparent, allowing it to be modified without requiring changes in the application or objects. This is important as dependencies can occur and change during the lifetime of an application, possibly in an unpredictable manner.

3. Configurable replica group views

In the following sections we shall describe in detail the new replica group views which we have introduced to enable applications to specify dependencies between objects.

3.1 Clustered replica groups

Whenever a replicated object is used within an atomic action, its group view information must be obtained from the naming service. If a client makes use of a number of replicated objects, either explicitly (e.g., through application related objects) or implicitly (e.g., through a composite object) each group view must be separately obtained, imposing an overhead on the application. In the situation where a composite object and (some of) its constituent objects reside within the same address space, it should be possible to make use of caching techniques to reduce the number of interactions with the naming service[3]. A *clustered replica group view* contains (references to) a number of individual group views, and whenever a single group view within the cluster is retrieved, *all* group views are retrieved (this is transparent to application users). This information is

2 Even if group view objects are not persistent and are stored efficiently by the naming service, e.g., with a hashed lookup scheme, scaling is still problem.

3 The techniques described here can also be applied to clients in different address spaces but for the sake of space we shall ignore these.

then cached, acting as a surrogate naming service for the clustered replica group and any other group view information which may be acquired during the atomic action. The cache is registered with the naming service to guarantee consistency between concurrent users, and is purged when the top-level action ends.

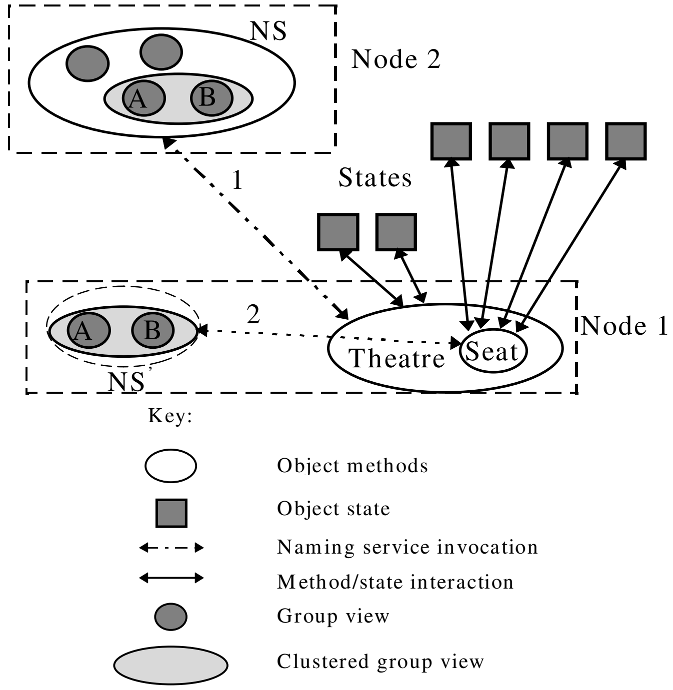

Figure 3: Clustered replica group views.

Figure 3 illustrates the use of a clustered group view for the *Theatre* and *Seat* composite object example. Each process is shown as executing on a different node and the (persistent) states of the *Theatre* and *Seat* objects are shown, and are assumed to also reside on different nodes. The naming service is shown containing four persistent (St_A) group view objects; for the sake of simplicity the states of these objects are not shown.

The naming service (*NS*) group views *A* and *B* represent the replication information for the *Theatre* and *Seat* respectively. Because there is an implicit dependency between these two objects (whenever the *Theatre* is activated, the *Seat* will typically be activated soon afterwards), the use of a clustered group view improves performance by reducing the number of remote naming service invocations. In this scheme, as soon as the *Theatre* object is activated within an atomic action and its group view is fetched from *NS* (message 1), the group view for the *Seat* is also returned. A surrogate naming service (*NS'*) is created within the *Theatre* process and initialised with this information. All subsequent naming service accesses (message 2) are indirected through *NS'*, and *NS* will only be contacted if the group view cannot be found (in which case it will then be cached within *NS'*).

3.1.1 Cluster managers

When a clustered group is created within the naming service it is available for other group views to be *registered* with it. Registration (and modification) occurs within an atomic action. The information about which groups are within a cluster is maintained within a separate *Cluster Manager Object*, allowing it to be inspected and modified independently, e.g., a management tool may inspect all clustered replica groups without acquiring locks on the group views or replicas mentioned. Information about which clusters a group is a member of is maintained within that group's group view information. This is necessary in order to be able to obtain the other cluster group views should a member view be explicitly requested.[4] This structuring of cluster information enables a user to request either the cluster view explicitly, or a group view for a constituent member and (transparently) obtain the entire cluster view.

Figure 3 illustrated the use of the surrogate (cached) naming service, which is created whenever a group view is obtained from the naming service. This cache can contain any number and type of group view from the naming service. Whenever a cluster view is obtained, the individual group views and their associated cluster manager are cached. When a group view is cached, a read lock is obtained on the group view within the main naming service; this lock exists for the duration of the cached copy. To guarantee consistency of group view information between concurrent users, a *write-through cache* is used, which must be manipulated within atomic actions, i.e., modifications to group views occur within the local cache and then the system attempts to impose these modifications at the main naming service (promoting the read lock held there to a write lock). If

4 A group can be a member of multiple clusters.

this update is not possible, for example because of a conflicts between users, the update action will abort, returning all of the group views (cached and original) to their original states [9]. Updating of the group views within the main naming service requires the acquisition of appropriate locks; to facilitate the simultaneous reconfiguration of multiple individual group views, when a lock is obtained on a clustered group view it is implicitly applied to all of the constituent group views.

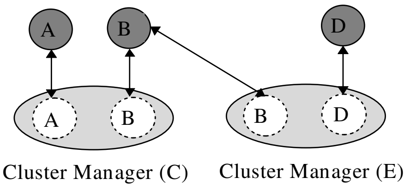

Figure 4: Cluster Management.

Figure 4 further illustrates this clustering technique. Group views *A* and *B* have been clustered together with cluster manager *C*, which is then represented as a group view within the naming service. If group view *A*, *B*, or *C* is requested then the entire cluster view will be returned. Figure 4 also shows how a group view can be clustered with a number of different cluster groups. Configuring a cluster requires its identity (e.g., the cluster group Uid) and the identity of the group to be added/removed. In the example above, if group view *A* is disassociated from the clustered group then the configuration information is (atomically) removed from both the cluster object and *A*. Likewise, if a new object (say, *D*) is added to the cluster, the clustering configuration information is added to both groups.

3.2 Template replica groups

When considering the replication of composite objects such as the *Theatre* and *Seat*, the replication information will typically be common. It was mentioned earlier that the performance of the naming service (i.e., the time taken to service a request) is dependent upon the number (and size) of group views registered with it. Therefore being able to reduce the number of group views can help improve performance. The introduction of the *template group view* solves this problem by factoring common information into a single object. As with the clustered group view, replica groups are then associated with this template; when a group view is requested, the naming service uses the template information and any other information present in the group view (e.g., the names of specific replicas), to obtain the final replica group view. The advantages of this scheme are:

- reconfigurations of multiple group views to change the number or location of replicas requires modification of only the template.

- reducing the amount of information stored within the naming service can improve its performance.

In addition to this *standard template* type, a special *wildcard template* can be added to the naming service which is used to generate a group view for an object which does not have a specific view registered with it. Using this template, it is possible to replicate every object. (Every group thus created will have the same number and location of replicas). If the naming service has this template as its only entry, then the entire object store can be easily replicated.

A surrogate (cached) naming service is also used by this group view mechanism to reduce the number of remote naming service invocations. However, since it is identical to that described in Section 3.1.1. it will not be mentioned further.

4. Implementation

We have implemented the model described in the previous section as part of the Arjuna distributed system [1][22]. Arjuna is an object-oriented programming system implemented in C++ that provides a set of tools for the construction of fault-tolerant distributed applications. Arjuna provides nested atomic actions for structuring application programs. Atomic actions control sequences of operations upon (local and remote) objects, which are instances of C++ classes. Operations upon remote objects are invoked through the use of remote procedure calls (RPCs).

Application classes obtain required properties through inheritance of an appropriate Arjuna class. State and lock management mechanisms are provided to user classes through the `LockManager` class, which in turn is derived from the `StateManager` class. Both classes are responsible for supporting the properties of serialisability, failure atomicity and permanence of effect of operations performed on user objects. The `StateManager` class provides facilities necessary for managing persistent and recoverable objects. These facilities include support for the activation and de-activation of objects, and state-based object recovery. If the object is replicated, it is the responsibility of `StateManager` to invoke appropriate group view operations on the naming service and contact the state replicas. If replica failures occur during update operations, StateManager excludes them from the replica group view. In addition to these classes, Arjuna also provides an AtomicAction class that can be used to create atomic actions.

The interface between user objects and the LockManager and StateManager classes is very simple, consisting of two constructors and seven operations. The two different constructors are used for creating either new objects or objects initialised with existing persistent object states. The StateManager class provides four virtual operations:

- save_state: to transform an object's current state into its passive representation, suitable for saving in the object store.

- restore_state: to transform an object's passive representation into its current state.

- type: used to locate the state in the object store.

- hasRemoteState: indicates whether or not the object is replicated.

StateManager also provides an operation get_uid which is used to obtain the Uid of the object. When the object is replicated, this is the replica state group Uid. For a new replicated object the operation setStoreInformation can be used to specify the number and location of replicas. The remaining operation, setlock, is provided by LockManager and is used by operations to indicate whether the object will be updated or just read.

To illustrate, we shall continue to use the theatre seat reservation example (originally taken from [21])[5]. The *Theatre* composite object controls booking access to its *Seat* constituent objects, allowing them to be atomically reserved or unreserved. The *Theatre* and *Seat* objects are replicated, derived from LockManager and possessing their own state management and concurrency control. The *Seat* class interface is given below:

```
enum SeatStatus { UnReserved, Reserved };

class Seat : public LockManager
{
public:
 Seat(char** nodes,int numberReplicas, Boolean& res);
 Seat(const Uid& objId, Boolean& res);

 Boolean Reserve   ();
 Boolean Unreserve ();

 // virtual LockManager methods

 Boolean save_state(ObjectState&);
 Boolean restore_state(ObjectState&);
 const TypeName type () const;
 Boolean hasRemoteState () const;

private:
 SeatStatus seat_status;
};
```

Figure 5: Interface to Seat class.

When creating a new *Seat* object, it is necessary to specify the replica configuration. This information is passed to the constructor by the nodes and numberReplicas parameters. The constructor uses setStoreInformation to pass this directly to the StateManager class. A new group view is created and registered within the naming service if the top-level action commits.

```
Seat::Seat (char** nodes, int numberReplicas, Boolean& res)
{
 AtomicAction A;

 res = FALSE;
 seat_status = UnReserved;

 A.Begin();  // start the atomic action

  // create the replica group

 setStoreInformation(nodes, numberReplicas);

 // commit action, implicit abort if failure
```

5 For simplicity, various error checking code has been omitted.

```
 if (A.End() == COMMITTED) res = TRUE;
}
```

Figure 6: New replicated Seat constructor.

4.1 Replica configuration classes

The interfaces to both of the replica configuration mechanisms described have been provided by two classes: `TemplateObjectControl` and `ClusterObjectControl`. These classes are the application's way of interacting with the naming service.

```
class TemplateObjectControl
{
public:
 TemplateObjectControl(const Uid&);
 ~TemplateObjectControl ();

 Boolean addTemplate (char** nodes, int number);
 Boolean deleteTemplate ();
 Boolean addToTemplate(char* node);
 Boolean deleteFromTemplate(char* node);
 Boolean groupWithTemplate(const StateManager& object);
 Boolean ungroupFromTemplate(const StateManager& object);
};
```

Figure 7: Template configuration class.

- `addTemplate`: creates a template group view within the naming service with the specified number and location of replicas; the group's identity is specified in the constructor.
- `deleteTemplate`: removes the template from the naming service.
- `addToTemplate`: adds another replica location to the template.
- `deleteFromTemplate`: removes the replica location from the template.
- `groupWithTemplate`: associates the user's object with the template.
- `ungroupFromTemplate`: disassociates the user's object from the template.

```
class ClusterObjectControl
{
public:
 ClusterObjectControl (const Uid&);
 ~ClusterObjectControl ();

 Boolean associateGroups  (const StateManager& object);
 Boolean disassociateGroups (const StateManager& object);
};
```

Figure 8: Cluster configuration class.

- `associateGroups`: adds the user's object to the cluster group view specified in the constructor.
- `disassociateGroups`: removes the user's object from the cluster specified in the constructor.

The following code illustrates the use of a cluster group view for the *Theatre* and *Seat* example, shown in Figure 3. In this example, the new *Theatre* object is created and information about its replica configuration is passed to `StateManager` via `setStoreInformation`. A new cluster replica group is then created within the naming service, the Uid of which is part of the *Theatre* state. The constructor then associates the *Theatre* object and all of the *Seat* objects which it creates with this cluster group.

```
class Theatre : public LockManager
{
public:
 Theatre (char** nodes,int numberReplicas, Boolean& res);
 Theatre (const Uid& objId, Boolean& res);

 Boolean Reserve  (int SeatNumber);
 Boolean Unreserve(int SeatNumber);

 // virtual LockManager methods

 Boolean save_state (ObjectState&);
 Boolean restore_state (ObjectState&);
 const TypeName type () const;
 Boolean hasRemoteState () const;

private:
 Seat* _seat[THEATRE_SIZE]; // theatre seats
 Uid   group;               // cluster identity
};
Theatre::Theatre(char** nodes, int numberReplicas, Boolean& res)
{
 AtomicAction A;

 res = FALSE;

 A.Begin();  // begin the atomic action

 setStoreInformation(nodes, numberReplicas);

 ClusterObjectControl control(group);
 control.associateGroups(*this);

 if (setlock(WRITE) == REFUSED)
 {
  A.Abort();  // abort the atomic action
  return;
 }

 for (int j = 0; (j < THEATRE_SIZE); j++)
 {
 _seat[j] = new Seat(nodes, numberReplicas, res);
  control.associateGroups(*_seat[j]);
 }

 if (A.End() == COMMITTED)
    res = TRUE;
}
```

Figure 9: Theatre and Seat cluster group.

4.2 Reconfigurations

An initial replica configuration for an object may require modification during the object's lifetime. For example, to continue to provide an availability level in the presence of increased component failures the number or location of replicas may need to be changed. In addition, changes in application requirements may affect the replication configuration. In Section 1 we described how certain executive theatre seats may have higher availability requirements than others. In the example code below, a new template group, with a different replication configuration to that of "standard" theatre seats, will specifically be created to govern these *Seat* objects.

We will assume that the *Theatre* class has the following additional method to accommodate this reconfiguration:

- `reconfigure`: given a new replication configuration (number and location of replicas) and a list of seats, this method creates a template group view and associates the specified seats with it. If successful, the seat replication configuration will be governed by the template.

The *Theatre* code performs all group view modifications within an atomic action. It removes each seat from the original cluster group and then associates it with the newly created template, which the replication sub-system will use to replicate the *Seat* object. The atomic action will only commit if every seat is successfully associated with the new template. Otherwise the action is aborted and none of the *Seat* replica configurations will have changed.

```
Boolean Theatre::reconfigure(char** nodes, int numberReplicas, int seats[], int numberSeats)
{
 AtomicAction A;
 Uid tmplId;      // template id

 A.Begin();
```

```
 ClusterObjectControl cCont(group);
 TemplateObjectControl tCont(tmplId);

 tCont.addTemplate(nodes, numberReplicas);

 if (setlock(READ) == REFUSED)
 {
 A.Abort();
 return FALSE;
 }

 for (int j = 0; j < numberSeats; j++)
 {
 if (!cCont.disassociateGroups(*_seat[seats[j]))
  {
   A.Abort();
   return FALSE;
  }

 tCont.groupWithTemplate(*_seat[seats[j]);
 }

 if (A.End() == COMMITTED)
    return TRUE;
 else
    return FALSE;
}
```

Figure 10 Seat replica reconfiguration.

4.3 Implementation results

To illustrate some of the advantages of using the replication schemes we have described, we have implemented the *Theatre* and *Seat* example. The implementation illustrates the use of a clustered replica group. The *Theatre* and each *Seat* object are independent Arjuna objects, possessing their own concurrency control and persistence mechanisms. Each object possesses three replicas, which are located in three persistent object stores, and each replica group view is independently registered with the naming service.

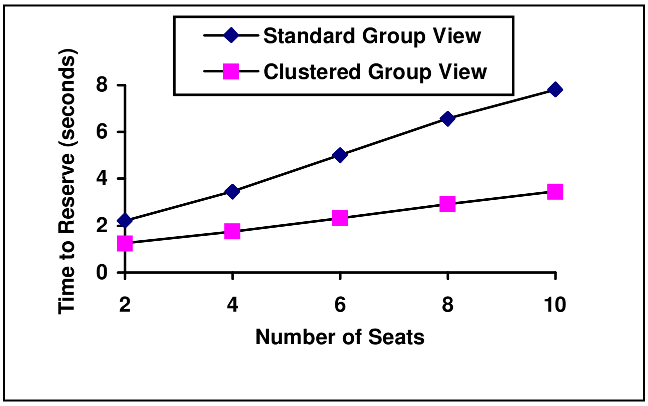

Figure 11: Reserving replicated theatre seats.

The graph shows the time taken for the `Reserve` operation for an increasing number of *Seat* objects. A `Reserve` operation involves: starting an atomic action; activating the *Theatre*; activating each of the *Seat* objects to be reserved; updating the state of each *Seat*; committing the action, which requires writing the modified object states back to each object store. For simplicity we consider the case where node failures and conflicts with other users do not occur.

In the first experiment no dependency information about the application and its objects is added to the naming service, and the *Theatre* and each *Seat* object must separately obtain their group views from the naming service. In the second experiment, dependency information is added to the naming service, and the *Theatre* and *Seat* group views are associated with each other in a cluster group view; as soon as the *Theatre* is activated, the group views for all of the *Seat* objects are also obtained and cached by the *Theatre*.

In both sets of results the time taken to complete the `Reserve` operation is proportional to the number of seats involved. However, in the first experiment, the requirement that all objects contact the naming service for their group views dominates the performance. In the second experiment, because the naming service is contacted only once (by the *Theatre*) and each subsequent access is fulfilled by the cache, the performance is improved.

5. Conclusions

We have shown that when replicating objects, by considering any inter-dependencies it is possible to obtain better performance and availability. To the best of our knowledge, no distributed system which supports replication provides the ability for applications to register this type of information. The replication sub-system we have presented maintains this information through the introduction of two new types of group view, the *cluster group view* and the *template group view*. The interfaces provided give users direct control over the configuration of a replicated object. Additionally, through the use of a special template, the *wildcard template*, it is possible to efficiently replicate every object in the distributed system. Thus, the mechanisms we have described scale well with the number of replicated objects. All of the work presented has been implemented within the Arjuna distributed system.

Acknowledgments

The work reported here has been partially funded by grants from the Engineering and Physical Sciences Research Council (EPSRC) and the UK Ministry of Defense (Grant Numbers GR/H81078 and GR/K34863) and GEC-Plessey Telecommunications.

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] B. Liskov, “Distributed Programming in Argus”, Communications of the ACM, Vol. 31, No. 3, March 1988, pp. 300-312.

[3] R. Balter et al, “Architecture and Implementation of Guide, an Object-Oriented Distributed System”, Computing Systems, 4(1), April 1991, pp. 31-67.

[4] M. C. Little, “Object Replication in a Distributed System”, PhD Thesis, University of Newcastle upon Tyne, September 1991. (ftp://arjuna.ncl.ac.uk/pub/Arjuna/Docs/Theses/TR-376-9-91_EuropeA4.tar.Z)

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman, “Concurrency Control and Recovery in Database Systems”, Addison-Wesley, 1987.

[6] D. L. McCue and M. C. Little, “Computing Replica Placement in Distributed Systems”, Proceedings of the 2nd IEEE Workshop on the Management of Replicated Data, Monterey, November 1992, pp. 58-61.

[7] M. C. Little and D. L. McCue, “The Replica Management System: a Scheme for Flexible and Dynamic Replication”, Proceedings of the 2nd International Workshop on Configurable Distributed Systems, Pittsburgh, March 1994, pp. 46-57.

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

[9] M. C. Little and S. K. Shrivastava, “Object Replication in Arjuna”, BROADCAST Project Technical Report No. 50, October 1994. (ftp://arjuna.ncl.ac.uk/pub/Arjuna/Docs/Papers/Object_Replication_in_Arjuna.ps.Z)

[10] L. Liang, et al, “Process Groups and Group Communications: Classifications and Requirements”, IEEE Computer, February 1990.

[11] M. H. Olsen, et al, “A Model for Interface Groups”, Proceedings of SRDS-10, Pisa, October 1991.

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

[13] E. Cooper, “Replicated distributed programs”, Proc. of 10th ACM Symposium on Operating System Principles, Washington, December 1985, pp. 63-78.

[14] K. Birman, T. Joseph and F. Schmuck, “ISIS - A Distributed Programming User's Guide and Reference Manual”, The ISIS Project, Department of Computer Science, Cornell University, Ithaca, NY, March 1988.

[15] 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.

[16] P. A. Alsberg and J. D. Day, “A Principle for Resilient Sharing of Distributed Resources”, Proceedings of the Second International Conference on Software Engineering, 1976.

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

[18] B. M. Oki, “Viewstamped Replication for Highly Available Distributed Systems”, PhD Thesis, MIT Laboratory for Computer Science, August 1988.

[19] D. K. Gifford, “Weighted Voting for Replicated Data”, 7th Symposium on Operating System Principles, December 1979.

[20] M. Ahamad, P. Dasgupta, R. J. LeBlanc and C. T. Wilkes, “Fault Tolerant Computing in Object Based Distributed Operating Systems”, Proceedings of the 6th Symposium on Reliability in Distributed Systems, March 1987.

[21] S. M. Wheater and S. K. Shrivastava, “Exercising Application Specific Run-time Control Over Clustering of Objects”, Proceedings of the 2nd International Workshop on Configurable Distributed Systems, Pittsburgh, March 1994, pp. 72-81.

[22] G.D. Parrington et al, “The Design and Implementation of Arjuna”, USENIX Computing Systems Journal, Vol. 8., No. 3, Summer 1995, pp. 253-306.

ERROR: undefined
OFFENDING COMMAND: D

STACK:

(n)
0
1.0
32
0