Object Replication in a Distributed System

by

Mark C Little

Ph.D. Thesis

September 1991

The University of Newcastle upon Tyne

Computing Laboratory

Abstract

A number of techniques have been proposed for the construction of fault–tolerant applications. One of these techniques is to replicate vital system resources so that if one copy fails sufficient copies may still remain operational to allow the application to continue to function. Interactions with replicated resources are inherently more complex than non–replicated interactions, and hence some form of replication transparency is necessary. This may be achieved by employing *replica consistency protocols* to mask replica failures and maintain consistency of state between functioning replicas.

To achieve consistency between replicas it is necessary to ensure that all replicas receive the same set of messages in the same order, despite failures at the senders and receivers. This can be accomplished by making use of order preserving *reliable communication* protocols. However, we shall show how it can be more efficient to use unordered reliable communication and to impose ordering at the application level, by making use of syntactic knowledge of the application.

This thesis develops techniques for replicating objects: in general this is harder than replicating data, as objects (which can contain data) can contain calls on other objects. Handling replicated objects is essentially the same as handling replicated computations, and presents more problems than simply replicating data. We shall use the concept of the object to provide transparent replication to users: a user will interact with only a single object interface which hides the fact that the object is actually replicated.

The main aspects of the replication scheme presented in this thesis have been fully implemented and tested. This includes the design and implementation of a replicated object invocation protocol and the algorithms which ensure that (replicated) atomic actions can manipulate replicated objects.

Acknowledgements

Firstly I would like to thank my supervisor, Professor Santosh Shrivastava, who suggested this area of research and with whom I have had numerous discussions over the years. I would also like to thank Dr. Graham Parrington and Dr. Stuart Wheater for reading and commenting upon the numerous drafts of this thesis. Their efforts are greatly appreciated.

I would like to thank my fellow members of the Computing Laboratory, in particular Dan McCue, Dr. Paul Ezhilchelvan and Xavier Rousset, for many useful comments and discussions I have had on this work.

Finally I would like to thank my family for their support and encouragement which they gave me during my studies. Without them it would not have been possible.

Financial support for much of the work described in this thesis was provided by a Research Studentship from the Science and Engineering Research Council and Esprit Project 2267 (Integrated Systems Architecture).

1: Introduction.

The increasing availability of cheaper, more powerful workstations and new networking capabilities has seen a corresponding increase in the use of distributed systems in many areas. These include applications in banking, office information systems, and real-time process control. Many organizations are by their very nature distributed, with individuals working in different locations but nevertheless requiring the ability to exchange information easily. Distributed computing systems have the potential for providing the facilities for cooperative work within such organizations.

Distributed systems also have the potential to be more extensible than centralized systems. For example, if an increase in the range of services is required (e.g., a dedicated processor could be added to a network which could be used by many people on the network) then it can be provided by adding the service and registering its presence so users can gain access to it. Although some centralized systems can also be extended in this way, distributed systems have the potential for much greater scaling.

Distributed processing also presents solutions to some of the problems associated with centralized systems when constructing reliable applications. Most notably, in a centralized computer system the probability that an individual component failure will cause the entire system to fail is high, leading to the situation where no services are available. Whereas in a distributed system it is possible to make use of the availability of resources on components with independent modes of failure to tolerate such failures.

However, because components are distributed (e.g., workstations separated by a communications medium) and the failure of one component does not automatically result in the failure of another component, this poses new problems not normally encountered in centralized systems. Failures such as message loss and corruption, processor crashes, and network partitioning, can create difficulties in maintaining the consistency of information stored over a number of components. Further, because of the increased number of components that make up a distributed system (machines, communication links, etc.)

there is a corresponding increase in the probability that one or more of the components can be faulty. Although these failures need not cause the entire system to fail, they do cause problems in maintaining the consistency of data and computations within the distributed system, leading to the necessity to create applications which are *dependable* despite faults in the distributed system.

1.1: Dependability and Fault Tolerance.

From the previous discussion it can be seen that it is essential to incorporate provisions for *dependability* in distributed systems if applications that can tolerate component failures are to be used. *Dependability* is defined as the trustworthiness of a component such that reliance can justifiably be placed on the *service* (the behaviour as perceived by a user) it delivers [Laprie 90][Anderson 81]. The *reliability* is a measure of continuous correct service delivery. A *system* is a collection of components connected by a communications medium, which can cooperate to perform some computation.

Keeping with the definitions in [Laprie 90], a *failure* occurs when the service provided by the system no longer complies with its specification. An *error* is that part of a system state which is liable to lead to failure, and a *fault* is defined as the cause of an error.

Since all physical components eventually fail, a computer system can be made dependable by making it *fault-tolerant*. A fault-tolerant system is one which is designed to fulfil its specified purpose despite the occurrence of component failures. Techniques for providing fault-tolerance usually require mechanisms for consistent state recovery and detecting errors produced by faulty components. The fault-tolerance technique that this thesis will concentrate on is the use of replication of resources on components with independent modes of failure to attempt to *mask* component failures in the system.

1.2: Replication.

Using *replication* it is possible to construct fault-tolerant services (i.e., services capable of tolerating component failures) by replicating vital system components (both in software

as well as hardware) and providing the notion of an *abstract component/service* to the users (one which exhibits the properties of a single component but is actually made up of many components). It will be assumed that replicated resources will reside on distinct nodes (workstations, or processors) in a distributed system (replicating on the same node has limited value, especially in terms of improving *availability* in the presence of node failures, as described below).

Replication can be used for two main reasons in a distributed system: increased availability, and increased performance. As we shall see, replication can also be used to provide a means of tolerating software design failures.

1.2.1: Increased Availability.

Consider a replicated service *S* and a user of this service *U*. When *U* issues requests on *S* this user may not know that the service is replicated and one reply from any replica in *S* may be sufficient for *U* to consider its request successfully executed. If a failure of one of the replicas which constitute *S* occurs, then *U* can still continue to operate. In this way the failures of replicas within *S* may go unnoticed until all of the replicas fail (this is the same behaviour as occurs within a non–replicated system if we consider the replicated service as a single logical service). The redundant components can be used to *mask* the failures of the other components from users of the service, with the number of replica failures that can be tolerated related to the number of correctly functioning replicas. A resilient service has the following requirements [Birman 85]:

1: Even after some maximum number of node failures, it must be possible to access a copy of the service.

2: Even after some maximum number of node failures, it must be possible to complete any operation in progress.

1.2.2: Increased Performance.

It is also possible to use replication to improve the performance of the distributed system. Consider the case of a single physical server component A and client components B and C. All client components which need to use A must communicate with it regardless of the load imposed on A. Furthermore, client B may have to wait for A to finish processing the request from client C before its request can be executed.

Now consider the case where A is made up of redundant components A_1 and A_2. All clients interact with this replicated service, but it may now be possible for B and C to use different replicas of this service, possibly also taking into account any network topology information to achieve better response times (e.g., be using the replica which resides nearest the client's node). If B and C request operations on A which can be performed concurrently (e.g., two read operations) then the performance of the system may be improved as each client's request is processed by a different replica without being delayed by the other's request. However, if the operations requested cannot be performed concurrently (e.g., modifying the same state) then they must be performed consistently by each replica. Typically this means that read operations are made to run faster at the expense of causing slower writes.

1.2.3: Design Diversity.

As long as each replicated object provides the same interface (set of operations) to a client, there is no reason why replicas should be physical copies of each other i.e., the same code. With different implementations of the same service (design diversity) it is possible that mistakes made in one implementation will not be made in another. Therefore a measure of tolerance against design faults can be obtained.

1.2.4: Replica Groups.

A natural way of managing replicas is to consider the individual replicas as members of a *replica group*. Providing and using replication then becomes a problem of managing the

interactions with groups rather than interactions with the individual group members (effectively changing the granularity from individual replicas to replica groups). The failure of a particular replica then becomes a separate issue to the failure of the entire group. Reliability of a service then depends on the reliability of the *group* providing the service.

When there is more than one replica in a group there is a possibility that different clients can make use of different replicas simultaneously, perhaps attempting to modify their states in a conflicting manner. A *replica consistency protocol* is necessary to ensure that concurrent invocations on different replicas leave all copies in a mutually consistent state. Such a protocol handles, among other things, concurrency control and recovery of failed replicas.

1.2.5: Replica Consistency Protocols.

There are two categories of replica consistency protocols which dictate the way in which replicas can interact with the rest of the system.

(i) active replicas – operations are invoked on all replicas of a given group, which execute the operations as though they were the only member of the group, and then return a reply to the invoker (client). If a client invokes an operation on a replica group with n members (assuming no failures) then the client will receive n identical replies to the original invocation.

(ii) passive replicas – operations are invoked on only one member of the replica group (the *primary*), which executes the operation and returns a reply. If a modification to the primary's state occurs then this primary checkpoints its state to the other members of the group (its backups). If the primary fails then the remaining members of the replica group will conduct an election amongst themselves to arrive at a new primary, which will then service subsequent requests.

Each of these approaches has its own advantages and disadvantages e.g., in passive replication checkpointing of state can be time consuming, while in active replication the problem of maintaining consistency between the replicas is harder because they all execute independently, which requires greater functionality from the communications medium. These issues will be described in more detail in chapter 3.

1.2.6: Further Aspects of Replication.

A replicated service will consume more system resources (network bandwidth, processor memory, etc.) than a non–replicated service. This increase in resource usage is offset by the improvement in the availability of the service. With extra modifications to the replication protocols and the underlying distributed system, it is possible to use these resources to provide other useful capabilities:

- *load sharing.* Replicas can cooperate to share the load (client requests) between themselves whilst at the same time maintaining consistency. It may be possible to improve further the performance of the replicated service by either increasing, or decreasing, the number of replicas as the application executes.

- *regeneration.* When a replica from a group fails it may be possible for the group to continue to function if the number of replicas within the group does not fall below a critical level (dictated by the replication protocol being used). However, if such failures are allowed to continue then the group will eventually fail completely. The regeneration mechanism causes the system to create new replicas if failures occur, instead of waiting for those failed replicas to recover. These replicas must be in a consistent state with any other replicas in the group so that an external observer (user) cannot distinguish between individual replicas.

- *replica migration.* If a client requires intensive use of a replicated service then it may be more efficient to move some of the replicas to the same node as the client. Replicas

would be moved across nodes to reside on or nearer to the node where the clients who are making requests on them currently reside.

1.3: Transparency.

The added complexities introduced by distributing and replicating system software and hardware components means that writing applications for such systems without any support is difficult. This has lead to the idea of *transparency*, which means that a distributed system can be made to behave, where necessary, like its non-distributed counterpart.

There are several complementary aspects to transparency [ANSA 89]:

- *access transparency* mechanisms provide a uniform means of invoking operations of both local and remote services, concealing any ensuing network related communications.
- *location transparency* mechanisms conceal the need to know the whereabouts of a particular service, making it sufficient to be able to name a service to access it.
- *migration transparency* mechanisms build upon the previous two to support movement of services from node to node for performance and fault-tolerance related reasons.
- *concurrency transparency* mechanisms ensure interference free access to services in the presence of concurrent invocations.
- *replication transparency* mechanisms increase the availability of services by replicating them but concealing the intricacies of replica consistency maintenance.
- *failure transparency* mechanisms help exploit the redundancy in the system to mask failures where possible and to effect recovery measures.

Before considering replication transparency in Chapter 3, we shall discuss in Chapter 2 some of the other tools and techniques used to build reliable applications, with regard to how they can aid in providing replication. In particular, the concepts of atomic actions

(concurrency transparency and failure transparency), remote procedure calls (access transparency), and object-oriented programming will be described as these are central to the replication protocols to be developed.

1.4: Contributions of the Thesis.

Data replication techniques have been extensively studied, especially for database systems. However, as we shall see, replicating objects is a much more complex problem because objects can encapsulate not only data, but also operations (methods) which act on that data. These methods can themselves contain calls on other objects. Thus, handling replicated objects is similar to handling replicated computations. The concept of active and passive replica groups will be discussed and it will be shown what advantages and disadvantages can be gained from each type: the problem of handling replicated computations is reduced if passive replication is used, but this then reduces the range of failures which the replicated service can tolerate.

When dealing with replicated objects, it is important to ensure that replicas within a group are consistent i.e., have the same state. Using active replication further complicates this as it requires co-ordinating interactions at each replica across the distributed system. This thesis presents the design and implementation of communication protocols for such co-ordination. In addition, we will describe how active replication protocols for objects were built for the Arjuna distributed system. We will show how in Arjuna it is not necessary to impose ordering of messages at the communication level but that such ordering can be imposed at the application level (we shall use atomic actions to do this). This thesis will further show how objects, which are used to encapsulate data and to hide internal implementation details, can also be used transparently to manage replica groups i.e., invocations on a replica (object) group will appear as though they are being made on a single object.

All of the main ideas developed in this thesis have been implemented and tested on a realistic distributed system (Arjuna). Where appropriate, we give performance figures for

our implementation. The software developed is expected to be used routinely in the Arjuna system.

We shall also show how message delivery guarantees cannot ensure replica consistency in situations where events occur locally to a replica and which cause the replica to take actions which may not be being taken by other members of the same group e.g., message buffers can overflow at a replica resulting in the loss of a correctly delivered message. This thesis will provide solutions to these problems, resulting in the design and implementation of a reliable group RPC mechanism. The replication protocols to be described and implemented will make use of this reliable group RPC.

1.5: Structure of the Thesis.

Relevant work on the construction of reliable applications in a distributed system will be discussed in the next chapter. Chapter 3 then describes the principles behind object replication. Chapter 4 describes the design and construction of a replicated procedure call mechanism, and shows how it is to be used in our replication protocol. Chapter 5 then discusses various replication protocols which have been built and used in other distributed systems, showing how they have approached the problems of replication. Chapter 6 then describes a new object replication protocol which has been designed and built for the Arjuna distributed system. The final chapter then gives the conclusions arising from the work discussed in the thesis.

2: Basic Fault-Tolerance Techniques.

In this section basic techniques which have been proposed for the construction of reliable applications will be described. These techniques are designed to provide solutions to the problems of inconsistency due to the partial failure of a distributed system, and also to the interference between users of an application. These issues, which are difficult to solve in an environment which does not make use of replication, are more difficult to solve when replication is used, as shall be shown in Chapter 3. One of the aims of this thesis is to show how such well understood mechanisms can be integrated with an appropriate replication protocol in a transparent manner, resulting in more reliable applications.

2.1: Object-Oriented Programming.

A programming methodology which is being increasingly used for software construction, is that of *object-oriented* programming. The object-oriented paradigm supports the concept of *data abstraction* (information hiding) which allows the programmer to associate a set of operations, which characterize the behaviour of the abstraction, with the data structures that represents the abstraction. *Objects* are instances of these *abstract types* (or *classes*). The objects can only be manipulated by the set of operations associated with them. Thus, as long as the actual interface remains the same, the implementation could be changed without affecting the user. If the language supports encapsulation, then the only way to access the data is by using the operations provided by the class.

A powerful concept provided in many object-oriented programming languages is that of *inheritance*. Inheritance allows the creation of a new class of objects which are either a refinement or embellishment on an existing class of objects.

There are many object-oriented programming languages, such as C++ [Stroustrup 86][Lippman 89] (which is based on the C programming language [Kernighan 78]), Smalltalk [Goldberg 83], Clu [Liskov 79], Trellis/Owl [Schaffert 86], Simula-67 [Dahl 70]

[Birwhistle 73] and Eiffel [Meyer 88]. However, throughout this thesis we shall concentrate on C++. An example for a *stack* object in C++ could be:

```
Class Stack
{
public:
      Stack();
      ~Stack();

      int push(int value);
      int pop(int& value);

private:
      int Elem[20];
      int top;
}
```

The private operations can only be accessed through the public interface, which in this case is the two operations `push` and `pop`, which perform operations on the private variables `Elem` and `top`. The two operations `Stack()` and `~Stack()` are special operations called the *constructor* and *destructor* respectively. The constructor is called automatically when an instance of this class is created (comes into scope) and can be used to call other operations (methods) necessary to initialise the object. The destructor is called automatically when an object goes out of scope and it deletes the object, freeing the memory it used, and can also perform other operations which may be necessary.

An object is termed a *persistent* (long-lived) entity if it continues to exist after the application which created it terminates. Nodes can have two types of storage: *volatile storage* (e.g., main memory), and *stable storage* (e.g., hard disk). Object states within volatile storage are lost if the node fails, whereas those within stable storage are able to survive node failures.

2.2: Atomic Actions.

One approach to making an object-oriented program fault tolerant essentially requires the objects themselves to be made fault tolerant. One method used to achieve this is to make use of the *atomic action* (or *atomic transaction*) [Gray 78][Lomet 77].

An *action* is a unit of work. During execution an action evolves as operations are performed on objects within the action's scope. An action appears to be indivisible to its surrounding environment i.e., it appears to be *atomic* to other actions. The properties of atomic actions are:

(i) serialisability – this property ensures that concurrently executing atomic actions appear to execute in a serial order i.e., they will be free from interference. To provide this property some form of concurrency control is required.

(ii) failure atomicity – a computation encompassed within an atomic action can be aborted without producing any effects. To achieve this property, objects must be made *recoverable* so that whenever an action is aborted the states which existed at the start of the action can be restored. This property requires *backward error recovery* and is invoked whenever a failure occurs which cannot be masked. Backward error recovery can be performed by either restoring the state of the object to that which existed at the beginning of the action, or by performing some compensating action which transforms the state to the state at the beginning of the action e.g., if the atomic action had incremented the object state by 5, the compensating action could decrease the state by 5.

(iii) permanence of effect – this guarantees that once an action *commits* normally (does not *abort*) the results produced are not destroyed by subsequent node crashes. This can be achieved if the object's state is recorded on non-volatile storage which can survive node crashes.

2.2.1: Action Primitives.

In any implementation of atomic actions there will typically be three basic primitives: *Begin_Action* and *Commit_Action*, which indicate the start and end of an action respectively, and finally the *Abort_Action* primitive, which can be used to abort the current action. Actions can also be nested within actions. If an action is nested, the enclosing

action (parent action) becomes the current action when the child action commits or aborts. The top-most action is called a top-level action.

```
Begin_Action(A)
      statements for action A              // A is the top-level action
      Begin_Action(B)
            statements for action B        // B is nested within A
      Commit_Action(B)
      possibly more statements for action A
Commit_Action(A)
```

Figure 2-1: Nested Atomic Actions.

2.2.2: Failure Atomicity.

To achieve the failure atomicity property objects can be made recoverable, so that if an action is aborted, the states of those objects which were modified within the action can be recovered to the states which existed at the start of the action. When an action commits, the states of those objects which were modified must be made permanent (*stable*) i.e., they are not lost due to subsequent node crashes. However, if an action is nested then the object states cannot be made stable until the outermost action (the *parent*) has committed (unless compensating actions are to be used), since this parent action (or other actions which enclose this committing action) could still abort.

The command *Begin_Action* represents the start of a *recovery region*. Whenever the *Abort_Action* primitive is called from within a recovery region it will cause all recoverable objects which were modified within the region to be returned to their earlier states. During the termination of a top-level atomic action, a *multi-phase commit protocol* is required to ensure that either all the objects which were updated within the action have their new states recorded on stable storage or no updates get recorded.

2.2.3: Concurrency in an object-based system.

In an object-based system the objects themselves can be made responsible for enforcing concurrency control. The most widely used form of such control is to regard

operations on an object to be one of two types: read or write. Concurrent reads are allowed since interference cannot occur, but if an action wishes to invoke an operation that would modify the state of the object it must have exclusive access to that object. This rule can be imposed by requiring that any action which wishes to perform an operation on an object must first acquire the correct type of *lock* for that operation.

To guarantee the serializability feature of atomic actions mentioned earlier, most systems which use atomic actions have adopted a *two-phase locking policy* [Eswaran 76][Moss 81] for the actions. Using such a scheme, no new locks can be acquired after any lock has been released, resulting in two distinct phases during the lifetime of an action: a *growing phase* where locks are being acquired, and a *shrinking phase* where locks are being released. The point where the atomic action has obtained all the locks it requires is called the *locked point*. Figure 2–2 illustrates the two-phase locking policy. The dotted line represents an atomic action using a *strict two-phase locking* scheme where all locks are released together, whereas the continuous line represents the general two-phase locking scheme.

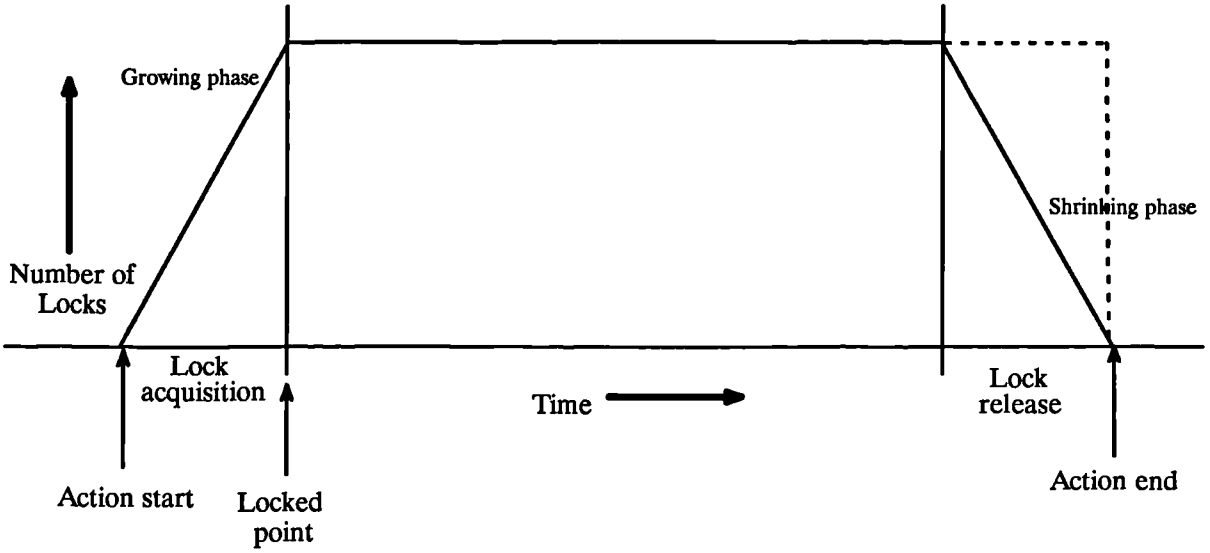


Figure 2–2: Two-phase locking scheme.

A method of providing increased concurrency between objects is to allow for *type specific concurrency control* [Schwarz 84]. The user can redefine the amount of

concurrency allowed within an object which will permit multiple clients to access the object. If the object provides a number of update operations which do not interfere with each other it should be possible to allow multiple update operations to be run concurrently instead of enforcing the "multiple-readers, single-writers" policy. An example of this would be the modification of two unrelated elements in a spreadsheet object, which requires write locks on these different elements.

2.2.4: Objects and Actions.

As has been shown in [LeBlanc 85][Shrivastava 88], the *object and action* model provides a natural way of structuring fault-tolerant systems with persistent objects. Persistent objects normally reside in *object stores* which are designed to be stable i.e., have a very low probability of failure. Atomic actions are used to control the state changes which occur on these objects, with the properties of atomic actions ensuring that only consistent changes can occur, despite failures.

An object normally resides in one object store. As shown in Section 1.2 to increase the availability of an object it is possible to replicate it on many different nodes. If a replication protocol is used in a system which also uses atomic actions then it is possible to integrate the two mechanisms. Atomic actions typically abort because remote services have failed; if these services are replicated then the probability that an action will have to abort as a result of such a failure is reduced.

2.3: Distributed Objects.

Since in a distributed system the objects within any given application may be located on physically distinct nodes, a communication protocol is required which will enable an application to invoke an operation on a remote object. Typically, this protocol will be implemented as a remote procedure call (RPC) mechanism that passes parameters from the calling application (client) to the server on the remote node which manages the object

the client wishes to invoke, and returns the result of the operation's execution. The RPC mechanism is a way of providing *access transparency.*

2.3.1: Remote Procedure Call.

The idea behind the *Remote Procedure Call* (RPC) [Birrell 84] is the fact that conventional procedure calls are well known and are a well understood mechanism for the transfer of data and control within a program running on a single processor. When a remote procedure is invoked, the calling process is suspended, any parameters are passed across the network to the node where the server resides, and then the desired procedure is executed. When the procedure finishes, any results are passed back to the calling process, where execution resumes as if returning from a local procedure call. Thus the RPC provides the system or application programmer a level of abstraction above the underlying message stream. Instead of sending and receiving messages, the programmer invokes remote procedures and receives return values.

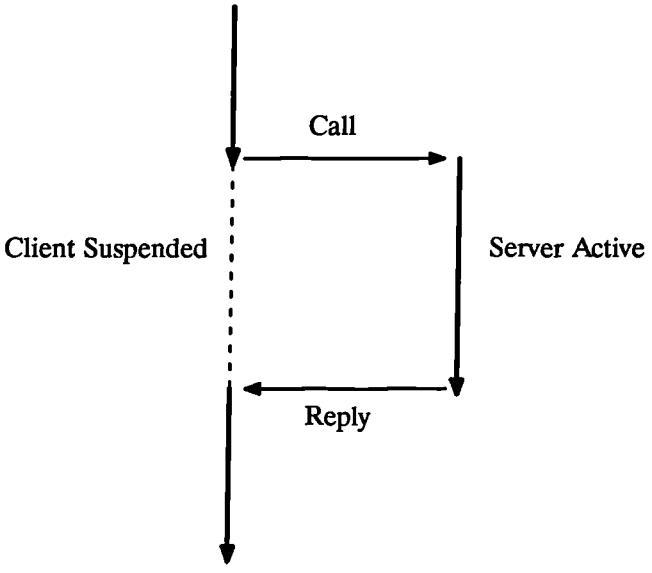


Figure 2–3: Remote Procedure Call.

Figure 2–3 shows a client and server interacting via a Remote Procedure Call interface. When the client makes the call it is suspended until the server has sent a reply. To prevent the sender being suspended indefinitely the call can have a *timeout value* associated with it: after this time limit has elapsed the call could be retried or the sender

could decide that the receiver has failed. Another method, which does not make use of timeouts in the manner described, instead relies on the sender and receiver transmitting additional *probe messages* which indicate that they are alive. As long as these messages are acknowledged then the original call can continue to be processed and the sender will continue to wait.

2.3.2: Groups.

[ANSA 90][ANSA 91a][Liang 90][Olsen 91] describe the general role of groups in a distributed system. Groups provide a convenient and natural way to structure applications into a set of members cooperating to provide a service. They can be used as a transparent way of providing fault tolerance using replication, and also as a way of dividing up a task to exploit parallelism.

A group is a composite of objects sharing common application semantics as well as the same group identifier (address). Each group is viewed as a single logical entity, without exposing its internal structure and interactions to users. If a user cannot distinguish the interaction with a group from the interaction with a single member of that group, then the group is said to be *fully transparent.*

Objects are generally grouped together for several reasons: abstracting the common characteristics of group members and the services they provide; encapsulating the internal state and hiding interactions among group members from the clients so as to provide a uniform interface (*group interface*) to the external world; using groups as building blocks to construct larger system objects. A group may be composed of many objects (which may themselves be groups), but users of the group see only the single group interface. [ANSA 90] refers to such a group as an *Interface Group.*

An *object group* is defined to be a collection of objects which are grouped together to provide a service (the notion of the abstract component discussed in Section 1.2) and accessible only through the group interface. An object group is composed of one or more group members whose individual object interfaces must conform to that of the group.

Interfaces are types, so that if an interface x has type X and an interface y has type Y, and X conforms to Y, then x can be used where y is used. This type conformance criteria is similar to that in Emerald [Black 86]. In the rest of this thesis, we shall assume for simplicity that a given object group is composed of objects which possess identical interfaces (although their internal implementations could be different).

The object group concept allows a service to be distributed transparently among a set of objects. Such a group could then be used to support replication to improve reliability of service (a *replica group*), or the objects could exploit parallelism by dividing tasks into parallel activities. Without the notion of the object group and the group interface through which all interactions take place, users of the group would have to implement their own protocols to ensure that interactions with the group members occur consistently e.g., to guarantee that each group member sees the same set of update requests.

By examining the different ways in which groups are required by different applications, it is possible to define certain requirements which are imposed on groups and the users of groups (e.g., whether collation of results is necessary from a group used for reliability purposes). [ANSA 91a] discusses the logical components which constitute a generic group, some of which may not be required by every group for every application. These components are:

- an arbiter, which controls the order in which messages are seen by group members.
- a distributor/collator, which collates messages going out of the group, and distributes messages coming into the group.
- member servers, which are the actual group members to which invocations are directed.

For some applications collation may not be necessary e.g., if it can be guaranteed that all members of a group will always respond with the same result. As we shall see in Section 2.3.3, if the communication primitives can guarantee certain delivery properties for

messages, then arbitration may also not be necessary. In general, all of these components constitute a group. In the rest of this thesis the logical components will not be mentioned explicitly, and the term *group member* will be used to mean a combination of these components.

2.3.3: Multicast Communication.

Conventional RPC communication is a *unicast* call since it involves one-to-one interaction between a single client and a single server. However, as shown in Section 1.2.4, when considering replication it is more natural to consider interactions with replica groups. *Group communication* is an access transparent way to communicate with the members of such a group. Such group communication is termed *multicasting* [Cheriton 85][Hughes 86].

Multicast communication schemes allow a client to send a message to multiple receivers simultaneously. The receivers are members of a group which the sender specifies as the destination of the message. A *broadcast* is the general case of a multicast whereby instead of specifying a subset of the receivers in the system every receiver is sent a copy.

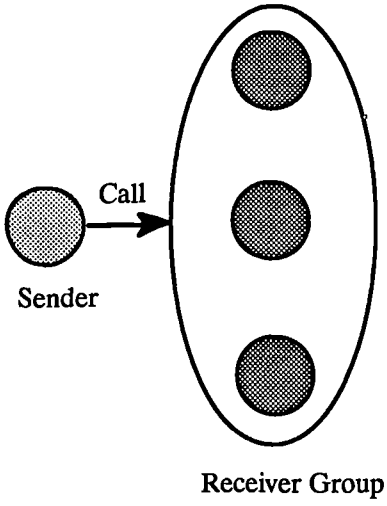


Figure 2-4: Multicast Communication.

Most multicast communication mechanisms are *unreliable* as they do not guarantee that delivery of a given message will occur even if the receiver is functioning correctly (e.g.,

the underlying communication medium could lose a message). When considering the interaction of client and replica group (or even replica group to replica group communication) such unreliable delivery can cause problems in maintaining consistency of state between the individual replicas, complicating the replication control protocol (if one replica fails to receive a given state-modifying request but continues to receive and respond to other requests, this resulting state divergence could result in inconsistencies at the clients). Thus, it is natural to consider such group-to-group communication to be carried out using *reliable* multicasts, which give certain guarantees about delivery in the presence of failures. These can include the guarantee that if a receiver is operational then the message will be delivered even if the sender fails during transmission, and that the only reason a destination will not receive a message is because that destination has failed. By using a reliable multicast communication protocol many of the problems posed by replicating services can be handled at this low level, simplifying the higher level replica consistency protocol. These protocols will be discussed in more detail in Chapter 4.

2.4: Summary.

In this section, techniques which have been proposed for building reliable applications were discussed. Object-oriented programming was described as a useful structuring methodology for building applications, and it was shown how the atomic action can be used to construct reliable applications.

Atomic actions provide an integrated mechanism which addresses the problems of inconsistencies due to partial failures of an application, and interference between concurrent parts of an application. The *failure atomicity* and *permanence of effect* properties of atomic actions ensures that if partial failures do occur then the application can recover to a consistent state, and the *serialisability* property ensures that operations which occur within concurrent actions and access the same objects can only occur in a consistent manner.

It was then shown how the RPC is typically used to enable remote objects to communicate with each other in a transparent manner. The functionality of the RPC matches closely that of the traditional procedure call, and as such, writing distributed applications is simpler because details of how messages are transmitted across the network are hidden from the application. However, the semantics of the conventional RPC implicitly assumes that communication will take place on a one-to-one basis (i.e., one transmitter and one receiver) and it was shown how there is the need for one-to-many and even many-to-many communication.

Groups were then described, introducing the concepts of the *object group* and the *replica group*, which showed what components are required within a group to enable it to function. It was shown that groups are not restricted to providing replication, but can be used in other areas. The functionality required from a group depends upon the application in which it will execute.

As soon as groups are used, one-to-many and/or many-to-many communication (*multicasting*) is required. Such group communication provides a transparent way to communicate with the many members of a group. Section 2.3.3 finished by describing why reliable communication is important when dealing with groups: so it can be guaranteed that every group member receives the same set of messages.

3: Principles of Object Replication.

By replicating resources on components with independent modes of failure, it is possible to construct fault-tolerant services, providing the notion of an *abstract component* to the users (one which exhibits the properties of a single component but is actually made up of many replicas). Each abstract component can be represented by a *replica group*, which is a number of replicas grouped together and cooperating to provide the same service. A replication protocol is required to manage all interactions with a replica group to preserve the notion of the abstract component despite failures in the components. The replication protocol must mask failures which occur so that the replicated service can continue to function. We shall see that there are a range of failures which can occur in a distributed system, and hence there is also a range of replication protocols which can tolerate these different failure types. The greater the range of failures which are to be tolerated, the more complex is the replication protocol required. We shall first examine the types of faults which can occur in a distributed system before discussing the replication protocols required to mask each failure type.

3.1: Replication and Failure Modes.

3.1.1: Failure Classification.

Given a (distributed) system, it would be useful if we were able to describe its behaviour formally in a way that will help establish the correctness of the applications run on it. If this then imposes restrictions on the permissible behaviour of the applications we will need to understand how those restrictions can be enforced and the implications in weakening or strengthening them. A useful method of building such a formal description with respect to fault-tolerance is to categorize the system components according to the types of faults they are assumed to exhibit.

Four possible classifications of failures are: omission, value, timing, and arbitrary. Associated with each component in the system will be a specification of its correct

behaviour for a given set of inputs. A *non-faulty* component will produce an output that is in accordance with this specification. The response from a faulty component need not be as specified i.e., it can be anything. As stated in [Shrivastava 90b] the response from a given component for a given input will be considered to be correct if not only the output value is correct but also that the output is produced on time i.e., produced within a specified time limit. In keeping with the definitions in [Shrivastava 90b] the classifications are:

- *Omission fault/failure*: a component that does not respond to an input from another component, and thereby fails by not producing the expected output is exhibiting an *omission fault* and the corresponding failure an *omission failure*. A communication link which occasionally loses messages is an example of a component suffering from an omission fault.

- *Value fault/failure*: a fault that causes a component to respond within the correct time interval but with an incorrect value is termed a value fault (with the corresponding failure called a *value failure*). A communication link which delivers corrupted messages on time suffers from a value fault.

- *Timing fault/failure*: a *timing fault* causes the component to respond with the correct value but outside the specified interval (either too soon, or too late). The corresponding failure is a *timing failure*. An overloaded processor which produces correct values but with an excessive delay suffers from a timing failure. Timing failures can only occur in systems which impose timing constraints on components.

- *Arbitrary fault/failure*: the previous failure classes have specified how a component can be considered to fail in either the value and time domain. It is possible for a component to fail in both the domains in a manner which is not covered by one of the previous classes. A failed component which produces such an output will be said to be exhibiting an *arbitrary failure* (*Byzantine failure*).

3.1.2: Fault Classification.

An arbitrary fault causes any violation of a component's specified behaviour. All other fault types preclude certain types of faulty behaviour, the omission fault type being the most restrictive. Thus the omission and arbitrary faults represent two ends of a fault classification spectrum, with the other fault types placed in between. The later failure classifications thus subsume the characteristics of those classes before them e.g., omission faults (failures) can be treated as a special case of value, and timing faults (failures). Such ordering can be represented as:

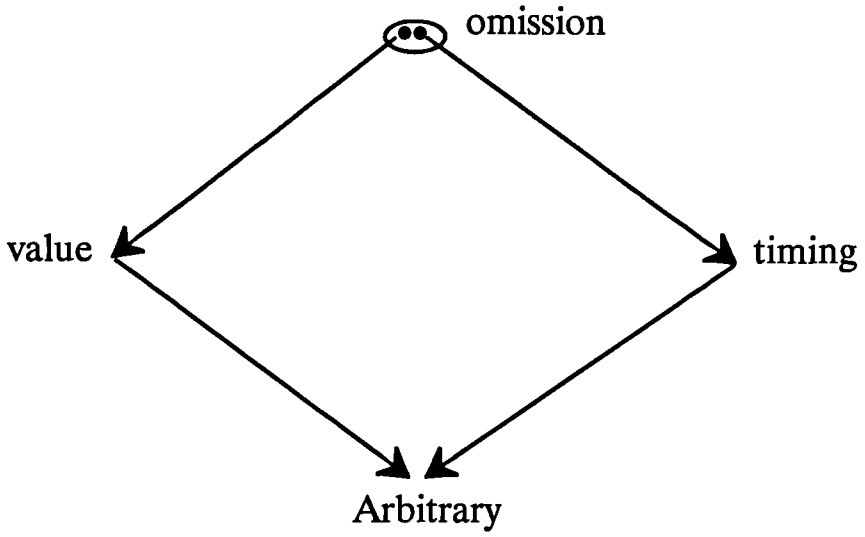


Figure 3–1: Failure Classification Hierarchy.

Now that we have examined the various classification of faults that can occur in a distributed system we shall discuss the classes of replication protocols which can be used to mask these faults should they occur.

3.2: Replication Overview.

The management of replicated objects is a complex operation. 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, method executions can result in 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 (Figure 3–2) which uses active replication, 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.

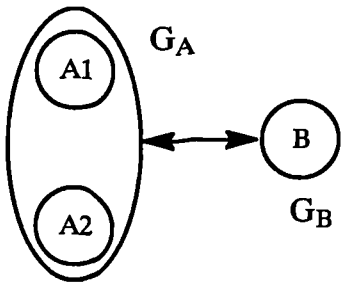


Figure 3–2: Object Groups.

Note that initially it will be implicitly assumed that all communication between clients and servers will be synchronous in nature i.e., a client is suspended until the server(s) has executed its request and returned a result.

3.3: Active Replication and Passive Replication.

It has already been mentioned in Chapter 1 that replication protocols break down into two classes: *active replication* and *passive replication*. Both active and passive replication can be used to mask failures of individual replicas within a replicated service and still enable the service to execute further requests from clients. However, the method of providing increased availability differs greatly between the two schemes. Because of the differences in the way individual replicas and replica groups are treated by these replication protocols it is not possible just to take a generic application and replicate it using either passive or active replication techniques. Each replication class implicitly assumes certain characteristics about both the application and the distributed system, which makes such general replication impossible practically. Note that although server groups will be discussed, it is possible for clients also to be replicated.

To be able to describe the way in which replication protocols function it is first necessary to make some assumptions about the failure modes of the components (nodes) upon which replicas will execute.

3.3.1: Fail-Silent Processors.

We shall initially assume that all nodes in the distributed system are *fail-silent*. A fail-silent processor is distinguished by its extremely simple failure-mode operating characteristics: it either works correctly or fails by halting without doing any spurious work. A fail-silent node exhibits *permanent omission* failures i.e., when a failure occurs the node stops functioning permanently (or until the fault can be corrected). Therefore a fail-silent processor never performs an erroneous state transformation due to failure: if a failure occurs, the processor simply halts.

A fail-silent processor is an idealized abstraction of real processors. However, given sufficient hardware it is possible to build realistic approximations to such processors [Schlichting 83][Shrivastava 90c][Barrett 90]. We shall see that when this premise is

relaxed, certain modifications need to be made to the replication protocols in order to compensate for the increased range of failures which can occur.

It is important to note that the internal state of a fail–silent processor and some predefined portion of the connected storage are assumed to be *volatile* (i.e., the contents are lost whenever a failure occurs). The remaining storage is defined to be *stable* (i.e., it is unaffected by any kind of failure). This stable storage property is necessary in order to continue a task that was running on a failed processor, since the state of that task must be available to the processor that is to continue it.

3.3.2: Passive Replication.

In a passive replica group (Fig. 3–3) only one member of the group (the *primary*) receives, evaluates, and responds to invocations from clients. To ensure that members stay mutually consistent the primary must send a checkpoint (snapshot) of its state to the passive members. In the event of failure of the primary, the remaining members use a protocol to elect a new primary, and this replica takes over from the failed primary and resumes execution of the operation from the most recent checkpoint. No further invocations can be processed until the new primary has been elected.

Since only one replica ever performs an operation, the replicas need not be deterministic in nature (something which, as we shall see, is necessary for active replication). Possible non–deterministic behaviour can be hidden by the fact that only the primary ever responds to the client and this response is the group response as the primary's state is also imposed onto its backups.

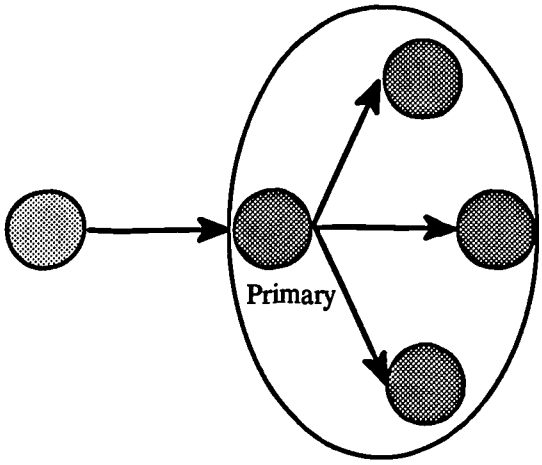


Figure 3–3: Passive Replication.

3.3.2.1: Determinism and Message Collation.

Because only one active member performs a given computation and then checkpoints its state to its backups, there is no need to ensure that the individual members of the replica group are *deterministic* (determinism means that given the same initial state and the same set of messages in the same relative order, all operational replicas will arrive at the same final state). As we shall see, if an object or application can possibly exhibit non–deterministic behaviour then this is the only protocol that can be used to replicate it.

Because a user of such a replica group only communicates with one member of the group this means that only one message typically passes between client and server group and the protocol to handle the reception of messages from replica groups can be simplified. (No form of collation of return results is necessary). The primary copy can receive multiple requests for the same operation from replicated clients (which may not be passively replicated), but each client should only receive one reply (from the primary), removing the need for a mechanism to handle multiple responses, as is necessary in active replication.

3.3.2.2: Primary Backups.

Backups which do not receive checkpoints for whatever reason, and hence are out-of-date, must be handled in such a manner that they cannot become the new active member of the group until they are consistent with the previous primary's state at its last checkpoint (which is the state that clients expect to find the replica group in). Ensuring this can take the form of the current primary excluding the unresponsive backups from the replica group until they have performed update actions. Another method would be for the primary election protocol to be sufficiently sophisticated that it cannot elect an out-of-date replica as the new primary without that replica first performing an update action. Either of these methods is sufficient to guarantee that the response a client receives is from an up-to-date primary. The updating of out-of-date backups can be performed by such a replica simply receiving the most recent checkpoint from the current primary.

3.3.2.3: Retained Results.

A potential race condition does exist if the primary fails after completing the execution of the client's request and checkpointing its state, but before it delivers the result to the client. The client may attempt to execute the request again, but this could cause problems if the operation is not idempotent (i.e., executing the operation twice does not yield the same result as executing it once). To avoid this, it is possible to enable the new active member to recognise the repeated request from the client as an operation which has already been performed and to simply retransmit the result again. ([Birman 85] referred to this as *retained results*).

For example, consider the case of a passive replica group being used to implement a cash dispenser. The current primary executes the request (R_1) to dispense the money, checkpoints its state (and the fact that it has executed R_1), and then crashes before replying to the client (the central banking computer, say). If the client retries R_1 the new primary will be able to determine that this operation has already been executed and instead of

dispensing the money again, simply transmits the acknowledgement to the client which the initial primary failed to do.

3.3.2.4: Failure Detection.

The need for a failure detection mechanism is obvious in this type of replication protocol. Such a mechanism regularly probes the primary member of the group to determine whether it is still operational (this can be done by sending "are you alive ?" messages and waiting for a response) and if a failure is detected then the backups are notified and the re–election process begins (it is assumed that failures can be detected eventually). The failure detection can be performed by the backups themselves, or by a separate, independent mechanism, or by the users of the group. However the failures are detected, the frequency with which the detector operates dictates how rapidly the replica group can recover from a failure of the primary, and hence how rapidly the group can resume executing requests on behalf of clients.

3.3.2.5: Primary Functionality.

From the initial description, it would appear that if all operations must be directed to the primary then this could become a bottleneck in terms of operation throughput. However, in some situations it may be possible for different replicas within the same replica group to become the active member for different clients as long as the operations they perform are not conflicting: for example, suppose that multiple clients wished to read or modify different, independent values in the same replicated object; in such a situation it would be possible for different server replicas to receive and service these requests as though they were the only primary.

When a client makes a request on the replica group it typically does so without having to know which replica is the current primary. The replication protocol guarantees that only one replica will ever respond to the client, and when this response is received it is guaranteed that this primary has checkpointed its state to a *sufficient* number of its functioning backups to be able to provide the level of fault–tolerance required.

The number of backups which must receive the checkpoint is application and system dependant. If the replicated service must be able to tolerate K node failures then K replicas must save the checkpoint as well as the primary. Checkpointing of state is an expensive operation which increases the time between a client issuing a request and receiving a reply. By reducing the rate at which checkpoints are made it is possible to improve the performance of the replica group when no primary failures occur, but at the expense of possibly having subsequent primaries re–execute a substantial amount of work, with subsequent effects on users of the replica group which may have to retransmit requests.

When a primary fails, users of a passive replica group may have to re–transmit requests which occurred after the last primary checkpoint, and therefore may also have to re–evaluate the replies from the new primary. As such, clients of a passively replicated service must maintain the ability to re–transmit requests and re–evaluate replies (rolling back their states to the state which they had when the failed primary last made its checkpoint). A checkpoint of the primary's state gives users of the group a point beyond which they will not have to rollback and can effectively "forget" about requests transmitted previous to this point. Note that information obtained from a passive replica group which has not checkpointed its state should not be transmitted to other objects until the checkpoint has been made, otherwise if the replica group fails then the number of objects which must undo operations can increase in a cascading manner.

Thus, there is a trade–off that must be examined on a per application basis, taking into account the needs of the application and the characteristics of the distributed system. For example, if the distributed system fails infrequently and read operations are performed more often than write operations then checkpointing of primary state can occur infrequently.

3.3.3: Active Replication.

In this protocol all members in an active replica group (Fig. 3–4) evaluate every invocation sent to the group and all members respond. Because all operational replicas execute the requests there is no need for checkpointing as is the case for a passive replica group. However, new members still need to be brought up–to–date with existing group members before they can be allowed to receive and respond to requests. When using an active replication protocol, there is an additional cost paid in the form of resources allocated to the redundant members which all need to execute. This is in contrast to the resource allocation needed for a passively replicated service, where typically only one replica executes requests.

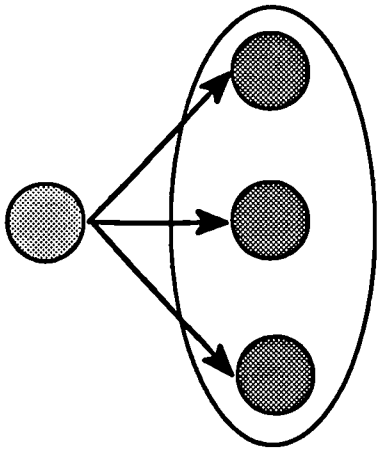


Figure 3–4: Active Replication.

Because every non–faulty member of an active replica group executes the same set of invocations it is necessary that such replicas be deterministic. If this were not the case then inconsistencies could arise in the replicated states. Also, since every non–faulty member of an active replica group responds to a request, some form of collation of these results is necessary at the client side. (If replicated clients exist then the collation problem is symmetrical at the server side). Principles of active replication have been developed using the *state machine* approach discussed below.

3.3.3.1: The State Machine.

In [Schneider 90] the *State Machine* was described as the general way to implement a fault–tolerant service by replicating services and co-ordinating the interactions of clients with such replicated services. This approach also provides a framework for understanding and designing replication management protocols which are based upon active replication. We shall use this framework here to show the conditions that must be satisfied for an active replication protocol to function correctly.

A state machine consists of *state variables*, which encode its state, and *commands*, which transform its state (thus a state machine can be thought of as being equivalent to an object). Each command is implemented by a *deterministic* program, and the execution of a given command is done atomically with respect to other commands and modifies the state variable and possibly produces some output. Requests are processed by a state machine one at a time, in an order that preserves any causal relationships between requests [Lamport 78]. Clients of a state machine can make the following assumptions about the order in which requests are processed:

O1: Requests issued by a single client to a given state machine *sm* are processed by *sm* in the order they were issued.

O2: If the fact that request r was made to a state machine *sm* by client c could have caused a request r' to be made by a client c' to *sm*, then *sm* processes r before r'.

Outputs from a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in the system.

3.3.3.2: State Machine and Fault–Tolerance.

A system consisting of a set of distinct components is *t fault–tolerant* if it satisfies its specification provided that no more than *t* components fail during some time interval of interest. A *t* fault–tolerant version of a state machine can be implemented by active replication of the state machine. Provided that each replica being run by a non–faulty

processor starts in the same initial state and executes the same requests in the same order, then each replica will perform the same set of operations and produce the same output. If the failure of a given processor (state machine) does not affect any of the other state machine replicas, then by combining the outputs of the remaining non-faulty state machines in the group, we obtain the output of the t fault-tolerant state machine.

When implementing an active replica group it is therefore necessary to ensure that all replicas receive and process the same sequences of requests. This can be decomposed into two requirements concerning the dissemination of information to the members of the group.

C1: every non-faulty state machine replica receives every request (*agreement requirement*).

C2: every non-faulty state machine replica processes the requests it receives in the same relative order (*order requirement*).

Requirement C1 governs the behaviour of a client which is interacting with state machine replicas. C2 governs the behaviour of a state machine replica with respect to requests from multiple clients.

There are various ways of implementing requirements C1 and C2, some of which will be described in later sections. However, these conditions do not make any assumptions about clients or state machine commands. With some knowledge of the commands it is possible to weaken the requirements, and allow cheaper protocols to be used. For example, requirement C1 can be relaxed for read-only requests when fail-silent processors are being used. When fail-silent processors are used, a request which does not modify the state variables need only be sent to one state machine replica. This is because the response from any one non-faulty replica is guaranteed to be the same as the response from any other non-faulty replica in the same group. Because the read-only operation does not change any state variables the one replica which does respond will still be consistent with those which do not.

However, the resource usage for an active replica group is greater than that for a passive replica group because each member of the group executes. The number of messages which flow across the communications medium is also greater, leading to the need for some form of collation of requests/replies. Optimizations exist which can reduce the resources usage and the number of messages, but these must be integrated into the replication protocol. For example, if each replicated server receives a copy of any reply that is sent by one of its replicas then it can use this information and decide not to send another copy of the same reply.

3.3.3.3: Operation Semantics.

Replication protocols typically rely on the ability of the invoking process to determine the type of operation that is being performed on the remote replica i.e., a read or a write operation. The protocols can function differently depending upon the operation type (e.g., in active replication it is only necessary for one replica to be contacted for a read operation, whereas all replicas must be contacted for a write). These are purely efficiency measures which can be made, as it is possible to assume every operation can modify the state of an object and design accordingly a less efficient, although still logically correct, replication protocol.

In an object–oriented system an object exports an interface with certain operations through which it is possible to manipulate its state. Some of these operations may modify the state, whilst others will leave it unchanged. From the exported interface definition it is not generally possible for a client to determine which operations modify the state and which do not.

This encapsulation of operations in the object makes the construction of a replication protocol which can exploit semantic information about the application and objects more difficult. The obvious solution to this is to take the pessimistic approach and assume every operation has the potential to modify the object state. However, this can impose substantial performance penalties since write operations are more expensive than read

operations. The other approach is to be optimistic and simply perform all operations on a single replica, with the understanding that if conflicts do occur some form of compensation (or rollback) must be carried out. These methods (and the one which we shall use, which allows the type of operation to be discerned) will be described in more detail in Chapter 5 and Chapter 6.

3.3.4: Communications Requirements.

3.3.4.1: Active Replication.

As was shown in Section 3.3.3.2, for active replication to function correctly there are two conditions which must be met (C1 and C2). This imposes certain restrictions on the way in which the underlying communications primitives can function. Both conditions can be met by making use of an *atomic broadcast mechanism*, which ensures that all functioning members of a given replica group will receive the same set of messages (condition C1) in the same relative order (condition C2), despite failures of the senders. If the communications medium cannot guarantee either of these conditions then further extensions to the replication protocol must be made to compensate e.g., only allowing operations to succeed if at least a majority of replicas receive and respond to requests (these modifications will be described in Section 3.4).

In Arjuna, which is an atomic action based system to be presented in Chapter 6, condition C1 is met by the communications layer, which uses a reliable broadcast mechanism which guarantees that despite sender failures all operation members of the same replica group will receive a given message. However, this does not ensure that the order of the received messages is the same between different replicas. Condition C2 is met by the higher level transaction serialization mechanism which imposes ordering only where necessary e.g., to ensure that conflicting operations are executed by all replicas in the same order. This separation of the conditions can typically improve the performance of the distributed application by allowing a simpler communications protocol to be used.

If asynchronous communication is used to allow client and server interaction, where the client is not suspended after issuing a request on a server and can collect replies at a later time, then it is necessary that the causal ordering between exchanged messages (the "happened before" ordering of messages), is preserved i.e., condition C2 should satisfy the causal ordering.

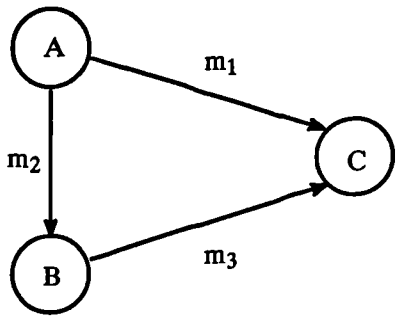


Figure 3–5: Exchange of Messages Requiring Causal Ordering.

Consider the situation shown in Figure 3–5. Client *A* issues an asynchronous request on server *C* (message m_1) and then issues another request on server *B* (message m_2). *B* then issues its own request on server *C* (message m_3), and this request arrives at *C* before the initial request from *A*. It would be incorrect of *C* to execute *m3* before *m1* as *m3* could depend upon work that should be done by *m1*. In such a situation *C* must be aware that there are other messages which should be processed first, which *m3* is dependant upon (i.e., the causal ordering). Condition O2 assumes that such causal orderings are preserved. Protocols which preserve the causal ordering of messages will be described in Chapter 4.

3.3.4.2: Passive Replication.

When using a passive replication protocol and communication is synchronous then client and primary interaction is similar to the more traditional client and non–replicated server interaction. Ideally the interaction between client and primary would be carried out with a communications protocol which at a minimum functionality ensures reliable delivery of messages. However, as in the non–replicated case, if an unreliable

communications protocol is used then messages could be lost between client and primary, but the client can simply retransmit requests until it receives a reply.

Communication between the primary and its backups must be atomic, however, since all of the operational backups must either receive a checkpoint or fail to receive it, ensuring only consistent backups are elected as new primaries. As stated previously, those backups which do not receive checkpoints must either be excluded from the group, or must update their states before they can be elected as a subsequent primary. If client and primary interaction is carried out asynchronously then all communication (including the checkpointing of primary state to the backups if multiple primaries are allowed to execute concurrently) must be at least causally ordered for the reasons discussed in Section 3.3.4.1. The ISIS system to be described in Section 5.3.1 provides a replication scheme which operates in the way.

3.3.5: Using Active Replicated Services.

Active replication is most appropriate in applications which require uninterrupted services, with minimum overhead during failures. In an environment which can satisfy the constraints outlined in Section 3.3.3.1, active replication has the promise of performing better than passive replication in the presence of failures, since there is no election of a primary or any need to ensure that backups are consistent. Although a service which is replicated using an active replication protocol consumes more resources than a similar service passively replicated, this increase in resources can be used advantageously. For these reasons active replication has been chosen as the main subject for this thesis. The advantages of this scheme are discussed below in detail.

3.3.5.1: Increased Performance.

Although replication can be used to improve the fault-tolerance of a given service, because multiple instances of this service must be contacted for each operation the overall performance of the service must be reduced (in some protocols read operations are made to run as though the service were not replicated, but write operations impose a greater

overhead than in the non–replicated case). In active replication, because conditions C1 and C2 must be met, this necessarily imposes an overhead on all client and replicated service interactions. In passive replication, the overhead comes about because the primary must checkpoint its state to its backups, waiting for sufficient replies before it can reply to the client (it proceeds at the speed of the slowest backup). However, it is possible to improve the performance of an application which uses an active replica group relative to one which uses a passive replica group, while at the same time satisfying the necessary conditions for it to function correctly.

Because there is no primary copy in active replication there is no need to run a (possibly) expensive re–election protocol as is the case for passive replication when failures occur. The replicas need not monitor the state (*liveness*) of other replicas in the group to determine when to run such a protocol either. Having to run such a protocol, as in the primary copy scheme, means that the replica group is unavailable while the protocol is taking place and clients must wait to have their requests processed i.e., the performance is degraded while the re–election takes place.

With an appropriate active replication protocol it may be possible for a client who makes a call on a replica group to continue to execute (stop waiting for more replies) as soon as the first reply is received from the fastest replica. In such a situation the performance would be almost the same as that in a non–replicated interaction and yet still provide the fault–tolerance and availability aspects. Also, it may be possible for a client to issue its requests on those replicas which are located physically close to it in the distributed system, hence reducing the overhead incurred by communicating over large distances of the network.

3.4: Replication and Failure Masking.

From the discussions of the two classifications of replication protocols it can be seen that there is a restricted subset of applications which can be replicated with each protocol type. This section will describe the type of faults that they can be used to mask. With a

restrictive fault model (i.e., assuming only a certain subset of faults which can occur) then the replication protocol can be simplified, and vice versa. The lattice of failure classifications outlined in Section 3.1.2 will also be followed, as any replication protocol which can function correctly in a less restrictive failure environment can continue to function correctly in a more restrictive failure environment.

We shall consider the number of replicas that are required by each replication protocol to be able to tolerate K replica failures, given a specific set of failure characteristics for the distributed system (assumptions about how the components will perform). We will assume that the only communication failure is that of network partitioning which prevents a functioning process from communicating with another functioning process. Other communications problems such as corrupted or duplicated messages can usually be detected and corrected for using well understood techniques used in most distributed systems. Aspects of the replication protocols concerned with replica recovery will not be considered here as they are orthogonal to the issue to be discussed (they will be discussed in Chapter 5).

3.4.1: Active Replication.

3.4.1.1: Permanent Omission Failures.

If we assume that the components in the distributed system either function correctly or cease to function in a silent manner (i.e., are *fail–silent*) then they can only suffer from permanent omission failures. In the absence of network partitions, K out of a total of K + 1 replica failures can be tolerated before an object becomes unavailable (which is termed *K–Resilient* [Birman 85]). Because components fail cleanly, the replication protocol can be simplified in terms of the state machine description of active replication: there is no need for a collation of return values from functioning replicas, as they will all be identical; a client need only read from a single replica as long as write operations are performed on all functioning replicas.

3.4.1.2: Value and Omission Failures.

If a failed replica can produce erroneous outputs (rather than no outputs) then it becomes necessary to validate those outputs from the replicas, by comparing the results produced. Note that a replica which suffers from occasional omission failures falls into this category as it may miss some update operations to its state and hence respond incorrectly to subsequent requests. To be able to tolerate K such failures it then becomes necessary to have $2K+1$ replicas in a group (so that a majority of correctly functioning replicas can always be obtained), and for the client to collate the replies from these replicas and perform a majority vote on them.

Network partitions can also be the cause of omission faults and so in order to be able to function in the presence of such partitions it is also necessary to have $2K+1$ replicas and the replication protocol must only allow operations to occur in the majority partition, if one exists.

3.4.1.3: Timing Failures.

Timing failures can only occur in those systems which impose timing constraints. If there are no constraints on "when" operations can occur or "when" results must be returned, then such messages can be neither too early, nor too late. However, most distributed systems and applications function by making use of such timing constraints, for example to detect when a process has crashed, or that a message has been lost.

Components which suffer from timing failures must be handled so that they appear as though they have crashed i.e., timing failures are masked to appear as permanent omission failures. In this way, only those "functioning" components (those replicas which respond in a timely manner) are ever read from and written to, in much the same way as happens for components which have crashed. This is because it is not possible (in some finite period of time) to distinguish a timing failure from an omission (crash) failure.

If timing failures occur it is necessary that all clients have the same view of which replicas have up–to–date states (i.e., have received all requests and responded on time) as

these are the only replicas which can be guaranteed to be correct at this time. To be able to tolerate K failures in such an environment, it is therefore necessary to have a minimum of 2K+1 replicas. The replication protocol must then collate the responses from the individual replicas and decide whether or not a given request can be considered successful, by only allowing operations to occur if a majority of the replicas respond on time with the same values. The protocol must ensure that despite components failing, multiple clients accessing the same replica group have a consistent view of the group and its state, and that conflicting operations cannot be performed. The Voting protocol to be described in Chapter 5 ensures this by only allowing operations to be performed if a specific subset (typically the majority) of the replicas can be contacted (called a Quorum). Although different clients may obtain replies from different subsets of the replicas in a given group, as long as the replicas which the clients contact intersect then each client will be able to observe the same state.

3.4.1.4: Arbitrary Failures.

If we assume that components can suffer from arbitrary failures (Byzantine failures) [Schneider 84] then a complex agreement protocol must be run before any actions can be performed on a replica group, or any results from such a group can be used. Each replica must exchange messages it has received with the other members of the group and they then perform a majority vote on the messages to determine what the final request/reply is. The messages exchanged between members must include those messages which were received from other replicas within the group as part of this agreement protocol.

As shown in [Lamport 82], in such a situation, ensuring consistency between replicas in the same group requires 2K+1 replicas if messages can be authenticated i.e., it is possible to determine that the message sent is correct and has not been corrupted at source (digital signatures can be used to implement authentication with a high probability). By performing a majority vote on the results obtained it is possible to use the majority value which will be correct provided no more than K failures occur.

If message authentication is not possible then 3K + 1 replicas are necessary to tolerate K failures because it is no longer possible to distinguish a message sent by a faulty replica from a correctly functioning replica which is simply retransmitting an incorrect message it received from the replica which is actually faulty.

3.4.2: Passive Replication.

3.4.2.1: Permanent Omission Failures.

As with active replication, if the components are fail-silent then it is possible to have a protocol which is K-resilient. Failures of backups have no affect on the service unless K such replicas fail making it impossible for the primary to checkpoint its state. Failure of the primary results in a new primary being elected from the remaining backups. If no backups remain then the replicated service has failed completely. Therefore the service can tolerate K replica failures out of a total of K + 1 replicas, whether the failed replicas are primaries or backups, before becoming unavailable.

3.4.2.2: Other Failures.

It is not possible to tolerate the other classes of failures with a passive replica group because only one member of such a group (the primary) transmits and receives messages. No collation of replica states is possible, as in active replication, to obtain a majority view. If the primary responds to a client then the result must be assumed to be correct (unless the message is corrupted, and can be detected as such). The primary imposes its view and its state on users of the group, and because such users never get to interact directly with the primary's backups these cannot be used to detect and mask other failure types as they are in active replication. As such, if the characteristics of the distributed system fail to meet the assumptions in Section 3.4.2.1 then passive replication cannot be used to replicate a service and ensure that the service continues to operate correctly in the presence of component failures.

It is however possible to tolerate network partitions in a passive replica group. The problem arises when a new primary has to be elected. Since it is not possible to

differentiate a failed replica from one which has been partitioned, it becomes possible for each replica group which spans a partition to have a separate primary. To prevent this requires $2K+1$ replicas and requires the primary election protocol to vote on whether a new primary should be elected. If a majority of the replicas can be contacted then a new primary can be elected, otherwise no new primary can take over.

3.5: Summary.

This chapter concentrated on the principles of replication, starting with a description of the classes of failures normally encountered in a distributed system and which replication of resources is typically used to mask. Then the requirements from a replication protocol were described, and it was shown that managing replicated objects is more complex than managing replicated data because an object's methods could contain calls on other objects. Hence, the problem of managing replicated objects amounts to being able to handle replicated computations, which can be best formulated in terms of managing *replica groups*. To avoid inconsistency it is necessary to ensure that a group appears to behave like a single entity in the presence of concurrent invocations and failures.

The two classifications of replication protocol were then discussed in detail: *active replication*, and *passive replication*. The differences between the two classes of replication were described, and it was shown that active replication required much more complex communications protocols than passive replication. Also, active replication assumes that all replicas are deterministic i.e., given the same starting state, and the same set of operations in the same order, they will all arrive at the same final state. Passive replication does not impose such a constraint. When considering active replication it is therefore necessary to use the State Machine approach. The two State Machine conditions were described, which state that each functioning replica must receive the same set of messages and executed them in the same order. These conditions must be met in order to ensure that all replicas within the same group have the same state.

Finally, we showed how active and passive replication can be used to mask the different classifications of failures described previously. Because of the way in which replication is handled by each class of protocol, it was shown that passive replication can be used to mask only a subset of the types of failures which active replication can mask. It was further shown how the number of replicas required to tolerate K failures increases as the severity of the failure model increases.

4: Replica Group Communication.

As has been shown in the Chapter 3, the functionality of the communications primitives of a distributed system and the characteristics of its communications medium must be taken into account when considering the interaction of replica groups. Whether these services are replicated actively or passively also affects what is required from the communication level e.g., collation of requests and results. It was also shown that the communications requirements for active replication can be much more demanding than those for passive replication. As this thesis is principally concerned with active replica groups we shall now discuss in detail what functionality is required from the underlying communications layer and how it affects the replication protocols which will run on it.

When considering active replication it is a necessary condition that all functioning replicas receive and process the same set of messages in the same order (this was shown in Section 3.3.3.1). Messages sent to replica groups should be delivered to all functioning members of the group. If the message delivery property cannot be met then it is possible for a process on a functioning processor occasionally to miss receiving messages directed to it. Such behaviour can create inconsistencies in replicated resources. Consider the interaction of replicas shown in Fig. 4–1.

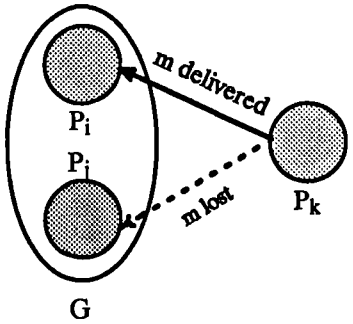


Figure 4–1: State divergence due to message loss.

Message m from P_k is being used by replicas P_i and P_j to determine whether that service is functioning or not. However, because the necessary message delivery property could not be met in this system, m is delivered to P_i but not to P_j, and hence P_j considers P_k to have

failed while P_i does not. P_j may then take some other action than that taken by P_i which may lead to its state diverging from that of its P_i.

Ensuring mutual consistency between members of the same replica group breaks down into two areas:

- maintaining replica consistency to *external events*: this ensures that all functioning replicas of a given group observe the same set of external events, despite failures. The external events considered are messages which are sent and received by objects. These messages must be ordered identically at each replica. Since typically clients and servers have associated message queues where messages are placed prior to being used, this can be obtained by having identically ordered message queues. However, this raises the problem of ensuring that the queues do not overflow.

- maintaining replica consistency to *internal events*: this ensures that all functioning replicas of a given group observe the same set of internal events, despite failures. The internal events which we shall consider are *local RPC timeouts* (on behalf of remote services). Because such timeouts can depend upon factors local to the client or server, such as the load on the host node or local network congestion, there can be no guarantee that the timeout has occurred at other replicas in the same group. Therefore, unless these events are handled consistently by each replica, the states of the replicas within a given group could diverge.

As was indicated in Section 2.3.1, a remote procedure call (RPC) can be terminated either when a timeout on behalf of the remote service occurs or when an "are you alive?" probe message fails to be acknowledged by the remote service. The problem of maintaining consistency to internal events as outlined above, only occurs if the RPC is implemented with timeout values. If probe messages are transmitted using communication protocols with some of the delivery characteristics to be described below, then they can be ordered consistently with respect to all other messages in the system so that they occur at every replica's message queue in the same order i.e., if one replica

determines that a server has failed, then all replicas in the same group will come to the same decision. Local timeouts cannot be so ordered.

For this reason, and the fact that the system that we designed our replication protocols for uses a timeout based RPC mechanism (to be described later), we shall consider only that type of RPC in what follows. Possible solutions to this problem will be described in Section 4.7. Note that active replication is also suitable in real-time environments where RPCs are often made with deadlines to ensure that actions are taken within a specified period of time. So our solution can be also extended to real-time systems.

We shall now discuss in detail the problems involved in maintaining replica consistency to external events, and propose some solutions by using the delivery guarantees that can be obtained from the communications layer. It is first necessary to consider the interactions which occur when non-replicated clients and servers interact across a network, before extending this reasoning to replica groups.

4.1: Remote Object Invocation.

Invocations on objects which are not replicated are traditionally based on the RPC as this retains the correct semantics of a procedure call i.e., a single flow (thread) of control from caller to callee and back again (as with a traditional procedure call). Section 2.3.1 described the concept of the Remote Procedure Call, and the simplified model of client-server interaction shown in Figure 4-2 will be assumed for the discussion to follow: a client uses the primitives *send_request()* for sending a call request and *receive_result()* for receiving the corresponding results. Clients and servers maintain enough state information to recognize and discard duplicate messages (filter requests). The server maintains a queue of messages from possibly multiple clients, and uses the primitive *receive_request()* to pick a message from the queue in a fifo order. After invoking the right method, the result is sent to the client with the *send_result()* primitive.

CLIENT

```
send_request()
repeat
      receive_result()
      filter_result
until valid_result
```

SERVER

```
repeat
      receive_request()
      filter_request
until valid_request
call object_method
send_result()
```

Figure 4–2: RPC Primitives.

When making replicated invocations (such as when calling a replica group) the semantics of such communication differ considerably from that of the traditional RPC: there is no longer a single thread of control, but rather multiple threads which may all eventually return to the caller. Such invocations are typically referred to as *Replicated Procedure Calls* [Cooper 84a][Cooper 85], and can be implemented using *one–to–many* (or *multicast*) communication facilities. We discuss various aspects of multicast communication below.

4.1.1: One–to–Many Communication.

The main services a multicast protocol provides can be categorised into three classes: *ordering, reliability* and *latency*. By imposing (increasing) ordering and reliability constraints on the delivery of multicast messages it is possible to define increasingly sophisticated protocols (typically at the expense of the latency). To understand these protocols first assume that a sender *S* is attempting to multicast to a group $G = \{P_1,...,P_n\}$. Following the definitions outlined in [Shrivastava 90b][ANSA 90]:

4.1.1.1: Unordered and Unreliable.

A multicast from S will be received by a subset of functioning nodes $Pi \in G$. Successive multicasts from S will be received in an arbitrary order at the destinations. Figure 4–3 shows sender S multicasting messages m_1 and m_2 to the group G. The first message is received by P_2 and P_n in different orders, and message m_2 is not received by P_1.

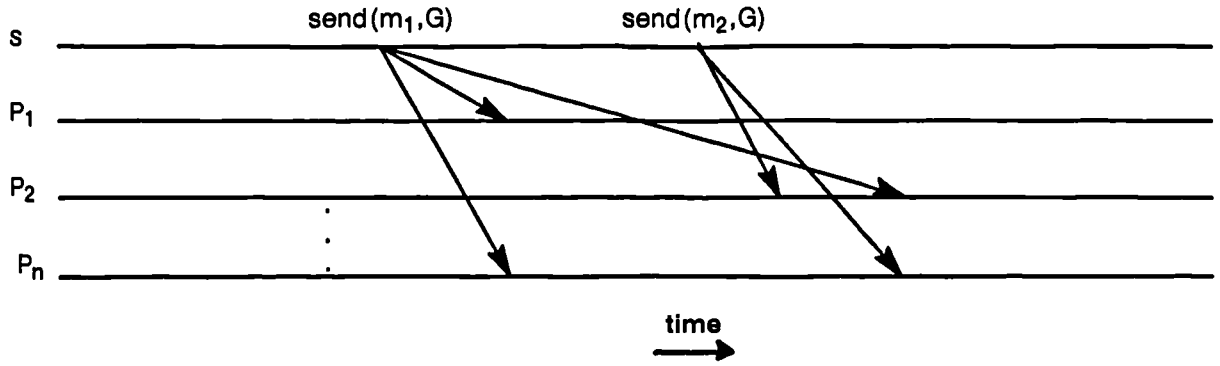


Figure 4–3: Unordered and Unreliable multicasts.

4.1.1.2: FIFO Multicast.

Provided the sender does not crash while transmitting the message, all correctly functioning receivers are guaranteed to get the message. Furthermore, the multicasts will be received in the order they were made.

Figure 4–4 shows two senders (S_1 and S_2) multicasting to the group G. All members of G received m_1 before m_2, but some members may receive m_3 before m_2. This last ordering is correct given the definition of the protocol: no information about the relative ordering of multicasts between senders is available to the receivers.

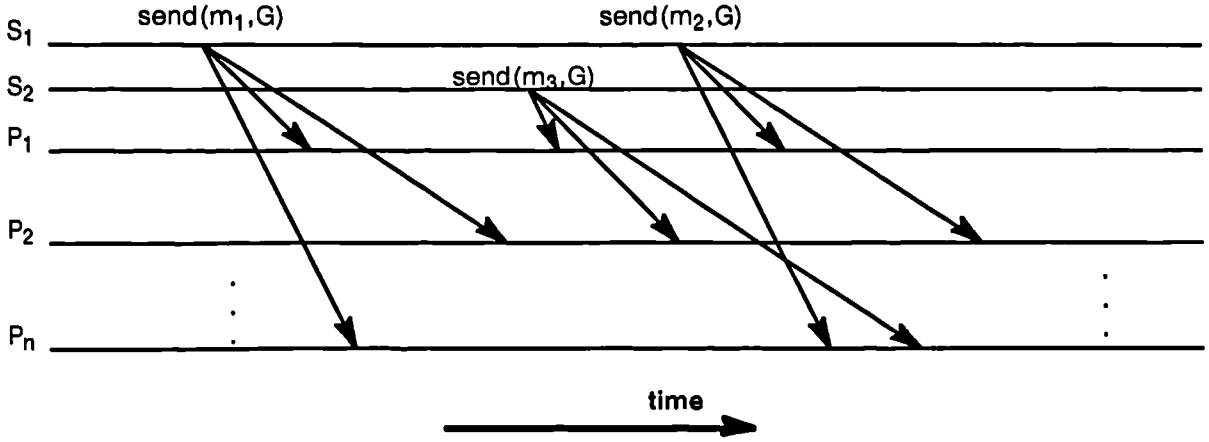


Figure 4–4: Ordered and Reliable multicasts.

4.1.1.3: Atomic multicast.

If the sender does not crash before completing a multicast, the message is guaranteed to be received by all functioning members. If, however, the sender crashes during a

multicast, then it is guaranteed that the message is received by either all or none of the functioning processes (*atomic delivery*). All multicasts from the same sender are received in the order they were made.

4.1.1.4: Causal multicast.

This multicast extends the ordering property of the Atomic multicast to causally related sends from different senders while still meeting the reliability guarantee. [Lamport 78] was the first to introduce the concept of potential causal relationships into computer interactions and showed what effects these relationships can have on the operations of interacting processes. Two events are potentially causally related if information from the first event *could have* reached the second event before it occurred. The notation used to denote such relationships is typically $X \rightarrow Y$, where $\rightarrow$ means *precedes* (happened before). Note that if X and Y are events from the same process and Y follows X then Y is necessarily causally related to X. A causal communication system will only preserve an ordering of events if the order is causally related. If two events are not related in this way then there is no guarantee on the delivery order.

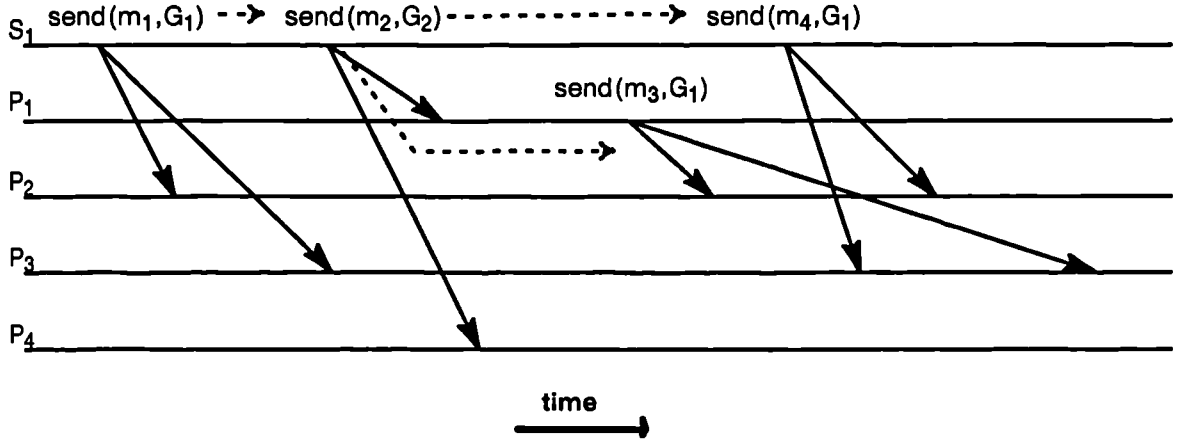


Figure 4–5: Causal multicasts.

In Figure 4–5, S_1 is multicasting the groups G_1 and G_2, P_1 is multicasting to group G_1. $G_1 = \{P_2, P_3\}$ and $G_2 = \{P_1, P_4\}$. There is a potential flow of information from send(m_1,G_1) to send(m_2,G_2), and from send(m_2,G_2) to send(m_3,G_1). This means that the sending of m_3 by P_1 is potentially causally related to the sending of m_1 by S_1. Hence the causal multicast

protocol must ensure that all functioning members of G_1 receive m_1 before m_3. Events such as m_3 and m_4 which are not causally related can be received in any order (they are termed *concurrent*).

4.1.1.5: Totally ordered multicast.

The (partial) causal order can be extended to a total order of messages such that all messages (whether causally related or not) are received by all destinations in the same order (which must also preserve causality).

4.2: Multicasts and Latency.

As described in [Shrivastava 90a], the *latency* of a multicast service is defined to be the time taken for a message, once sent, to reach the destination processes. This latency is particularly important for protocols providing reliability and ordering guarantees. As we shall see, whereas the latency for an unreliable multicast service is bounded (typically of the order of a few milliseconds), the latency for a multicast service which operates in the presence of failures (message losses and node crashes) can be bounded or unbounded depending upon the implementation.

Existing order preserving protocols can be broadly classified in the following way:

- *message history based*: the main idea behind such protocols is that when a process sends a message it appends some historical information about the messages it has received in its recent past. This historical information enables the receivers to retrieve any missing messages and to order them properly. This type of protocol ensures that an incomplete multicast is *eventually* completed, and hence possesses an unbounded latency property.
- *centralised distributors*: here the sender delivers the message to a specific member of the group (the primary) who is then responsible for distributing the message to the fellow members of the group. The primary can assign a total order to the messages it receives. As we have already seen, failure detection mechanisms are necessary to detect failed primaries and to elect new primaries which can take over and complete

the multicasts. Such protocols can possess bounded latency, but the necessity to detect asynchronously occurring failures can impose an overhead on performance.

- *multi–phase commit*: these protocols, providing total order, use multiphase algorithms (typically 2 or 3 message rounds) which are similar to the two–phase algorithm described earlier for atomic action commits. The sender delivers the message to the destinations which return sufficient information to the sender about the messages that they have received so that in the subsequent rounds of the protocol the sender can impose an identical order of the message onto the destinations. The message is only considered to have been delivered if all of the phases of the protocol complete. Such protocols provide bounded latency.

- *clock–based*: these protocols are an important class of the multi–phase algorithms, and assume the existence of a global time base. Timestamps derived from such a time base can then be used for imposing a total order on messages. Such protocols can provide constant latency communication, having the attractive property that if a sender multicasts a message at clock time T, then it can be sure that all functioning receivers will have received the message by clock time $T + \Delta$, where Δ is the constant indicating the protocol latency (Δ must be determined by applying worst case timing and failure assumptions).

We shall now examine a system which provides a reliable multicast protocol using a message history based approach.

4.3: Review of an Existing Multicast Protocol.

We shall now consider one implementation of a communication subsystem which provides some of the delivery properties described previously. It is important to understand how these ordering requirements can be met, and the overhead which is involved in guaranteeing them, before we discuss how such communication primitives can be used to provide replicated object groups. Other reliable communication subsystems

exist, of which [Chang 84][Cristian 85][Cristian 90][Verissimo 89] are a sample, but we shall consider *Psync* because it illustrates many points clearly.

4.3.1: Psync.

Psync [Peterson 87][Mishra 89] ("pseudosynchronous") is a communication subsystem designed to provide reliable multicast communication between objects, and is based on the message history approach described above. The system assumes that operations on objects which change the state occur atomically and are idempotent. Associated with each object is a manager process. A client process locates a particular manager (perhaps by consulting a naming service) and then invokes operations on the object by sending requests to that manager. When a manager receives a request to invoke a particular operation on an object, it encapsulates the operation in a message and uses the *Psync* many–to–many communications protocol to forward the message to all of the managers involved (including itself) if the object is a member of a group. Based on the set of received messages, each manager can then decide on an order in which to apply the operations to its copy of the object. This protocol can be extended to be used for the interactions of replicated object groups, and the exact details of the replication protocol used in *Psync* will be described in Section 5.3.5.

4.3.1.1: Conversations and Context Graphs.

Psync explicitly preserves the partial ordering of messages exchanged among a collection of processes in the presence of communication and processor failures (*Psync* cannot function in the presence of network partitions). A collection of processes exchange messages through a *conversation* abstraction. This conversation is defined by a directed acyclic graph (a *context graph*) that preserves the partial order of the exchanged messages. This ordering is made available to all managers involved in a conversation and by using this they can determine when to execute operations on their local objects.

When processes communicate they do so by sending messages in the *context* of those messages they have already received. Participants are able to receive all messages sent by

the other participants in the conversation but they do not receive the messages that they themselves send. Each participant in a conversation has a *view* of the context graph that corresponds to those messages it has sent or received. The semantics of the communications primitives provided by *Psync* are defined in terms of the context graph and a participant's view.

Figure 4–6 shows an example of a context graph. This conversation is started with the initial message m_1. Messages m_2 and m_3 were sent by processes that had received m_1, but are independent of each other (hence no link between them), and m_4 was sent by a process that had received m_1 and m_3, but not m_2. Messages that are not in the context of some other message are said to be concurrent (occur at the same *logical time*). The relation $<$ is defined to be "in the context of".

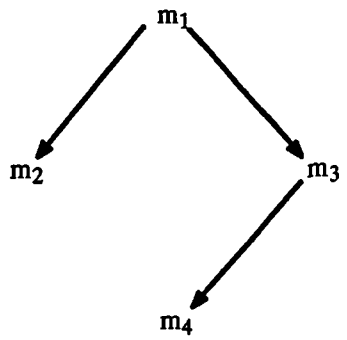


Figure 4–6: A Context Graph.

The context graph contains information about which processes have received what messages. The receipt of a message implies that the sender has seen all of the messages which came before it in the context graph. A message m_p sent by process p is said to be *stable* if for each participant in the conversation $q \neq p$, there exists vertex m_q in the context graph sent by q, such that $m_p < m_q$. For a message to be stable means that all participants except the sender have received it, it must follow that all future messages sent to the conversation must be in the context of the stable message.

In figure 4–7, we have a context graph which depicts a conversation between three participants, *a*, *b*, and *c*. Messages a_1, a_2, ... denotes the sequence of messages sent by process *a*, and so on. Message a_1, b_1, and c_1 are the only stable messages; messages a_2 and a_3 are two unstable messages sent by process *a*.

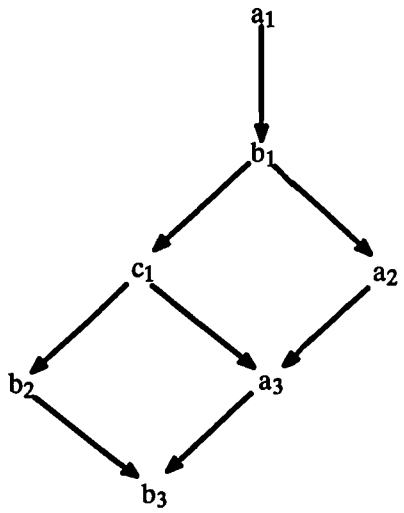


Figure 4–7: Stable and Unstable messages.

Psync maintains a copy of a conversation's context graph at each host on which a participant in the conversation resides. Each process in the conversation receives messages from this local copy of the context graph, which is termed the *image*. Whenever a process at one host sends a message, *Psync* propagates a copy of the message to each of the hosts in the conversation. This message contains information about all messages upon which the new message depends, so that the receiving hosts can append it to their context graphs in the correct place.

4.3.1.2: Dealing with Network and Host Failures.

Suppose message *m* is not delivered to some host because of a network failure. If at some future time a message *m'* arrives that depends on *m* then the host will detect that it is missing *m* and will send a retransmission request for *m* to the host that sent *m'*, (this host is

guaranteed to have *m* as a local participant has just sent a message which is in the context of it).

The operations provided to aid applications in recovering from host failures include the ability for a participant to remove a failed participant from its definition of the participant set for a conversation. This is necessary so that messages will eventually stabilize relative to the functioning participants. Once a given participant has been masked out, *Psync* ignores all further messages from that process.

There is also an inverse operation that allows a participant to rejoin a participant set. It would be invoked when a participant becomes aware that another participant which was formally failed has now recovered.

When a host recovers, a participant can initiate a recovery action which will inform other participants that the invoking participant has restarted, and to initiate reconstruction of the local image of the context graph. Each member of the conversation will transmit its local copy of the context graph to the recovering participant who can then use this to reconstruct its own local copy.

4.3.1.3: Total Ordering.

As described, the *Psync* protocol only gives a partial ordering of messages i.e., only the causal ordering of messages is preserved. To convert a partial order into a total order, whereby messages which are not causally related are ordered identically at all overlapping destinations, requires additional information to be shared amongst the destinations which indicates the order in which to place such messages. In *Psync*, the context graph which accompanies each message provides this information. The partial order that *Psync* provides can be used to give a total order if all participants perform the same topological sort of the context graph. This sort must be incremental i.e., each process waits for a portion of its view to be stabilized before allowing the sort to proceed. This is done to ensure that no future messages sent to the conversation will invalidate the total ordering.

The replication protocol used in *Psync* uses just such a scheme and will be described in Section 5.3.5.

4.4: Multicasts and Replication.

In the previous sections we have discussed what delivery properties can be provided by the communication subsystem, and how they can be implemented. We shall now discuss how such communications primitives can be used in a replication scheme to aid in maintaining replica consistency.

Group invocations can be implemented as replicated RPCs by replacing the one-to-one communication of *send_request* and *send_result* in Figure 4–2 with one-to-many communication. Every non-faulty member of a client group sends the request message to every non-faulty member of the server group, which in turn send a reply back. If multiple client groups invoke methods on the same replicated server group then it must be ensured that concurrent invocations are executed in an identical order at all of the correctly functioning replicas, otherwise the states of the replicas may diverge. In order to ensure this property, the objects must not only be deterministic in nature, but all correctly functioning replicas must receive the same sets of messages in the same order i.e., a totally ordered multicast must be employed.

The State Machine conditions C1 and C2 can both be met by making use of totally ordered multicasts to deliver all messages transmitted by clients and servers. However, such total ordering of messages may be unnecessary for all interactions: if two, non-conflicting, non-related messages are received at members of the same replica group (e.g., two unrelated electronic mail messages from different users) then they need not be ordered consistently at these destinations. If they were related in some manner (e.g., from the same user) then they could be ordered consistently. Application level ordering can be achieved more efficiently as cheaper, reliable broadcast protocols can be used to deliver messages for which ordering is unimportant, resorting to the more complex order preserving protocols only where necessary. Since such protocols typically need more

rounds of messages to be transmitted, the reduction in their use can be beneficial to the system as a whole, whilst maintaining overall replica consistency.

One method of achieving such application level ordering would be to transmit messages using *unordered atomic multicasts* (since it is still important that the messages are received by all functioning replicas) which only guarantee delivery to all functioning replicas but make no guarantee of the order (described in Section 4.1.1.3), and then to impose ordering on top of this i.e., at a level above the communication layer. If atomic actions are used in the system then we can make use of their properties to impose the ordering on message execution that we require i.e., the order shall be equivalent to the serialization order imposed by atomic actions. This has the advantage that operations from different clients which do not conflict (e.g., multiple read operations) can be executed in a different order at each replica. Atomic actions will ensure that multiple accesses from different clients to the same resource will be allowed only if such interaction is serialisable.

It is just such an approach that we shall describe in this thesis, starting with a description of the communication subsystem which has been implemented, called rel/REL. It provides atomic multicast functionality as described in Section 4.1.1.3.

4.5: The rel/REL Family of Multicast Protocols.

Using the failure model described in Section 3.3.1, we shall now describe the design of a reliable (unordered) atomic multicast protocol, which shall be used in the replication protocol to be described in Chapter 6.

If a client process α wishes to send a multicast message *m* to a server group G, then the delivery mechanism is said to be *atomic* if a functioning process $P_i \in G$ receives the message m and all functioning processes $P_j \in G$ will also receive *m* even if the sender fails during the call.

The message delivery property can be met realistically in bounded time 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.

The rel/REL reliable atomic multicast [Shrivastava 90a] was designed and built as a result of the work towards providing replication for the Arjuna system [Shrivastava 88]. As such, the replication protocol presented in Section 6.3 uses this multicast, although any other reliable atomic multicast could be used. rel/REL is actually a family of multicast protocols which can provide such functionality as causal ordering, total ordering, and atomic multicasting. The description we shall give will be that of the basic protocol rel/REL_{atomic} which provides reliable atomic multicasting, preserving order only between messages sent by the same sender, and follows closely the description given in [Shrivastava 90a].

The name results from the way in which the protocol works. Message transmission is carried out in two rounds: the first round (rel) is the transmission of the message in a reliable manner (as described in Section 4.1.1.2). This ensures that as long as the sender does not fail, all functioning destinations will receive the message. The second round (REL) is used to ensure that such guaranteed delivery occurs despite sender failure.

4.5.1: The rel/REL_{atomic} Protocol.

The protocol is developed in two layers: a lower level transport service, *rel*, and the next layer which provides fifo multicasts.

rel: *rel* provides a multicast transport service for one–to–many communication; its functionality is *fifo reliable (non–atomic)*. There could be several possible network specific

protocols for providing this service. For example, on a broadcast network such as an Ethernet, a multicast datagram service (unordered and unreliable) combined with acknowledgements and selective retransmissions could form the basis of implementing *rel*. In what follows, the existence of *rel* will be assumed.

Every host is assumed to have a TRANSMITTER process to which local processes wishing to perform multicasts send their messages via FIFO queues. The TRANSMITTER process has a number of concurrent threads (lightweight processes), say n, which are cyclically picking up messages from these queues (operation get_message(m)) and invoking a procedure *REL* for network transmission.

```
TRANSMITTER:
loop_forever do
{
      get_message(m);
      REL(m);
}
```

A message contains a list of destination addresses and a type field indicating which round of the protocol it was sent in (*type* = *first:* first round of the protocol, *type* = *second*: second round). The algorithm for the procedure *REL* is given below:

```
procedure REL (m : message)
{
            m.type := first; rel (m);  /* first multicast send...*/
            m.type := second; rel (m); /* and the second one */
}
```

Every host also has a RECEIVER process which is responsible for picking up messages. TRANSMITTER and RECEIVER processes use the services of *rel* for message transmission and reception.

RECEIVER:

```
loop_forever do
{
      receive(m);          /* receive message from network */
      case m.type of {
            first: if duplicate(m) then discard(m);
                  else {
                        deposit(m, m.dest); /* put m in queues of */
                                            /* m.dest */
                        m_thread = start_thread(m);
                        deposit(m, m_thread);
                        /* start a thread to finish protocol and */
                        /* deposit message in its queue */
                  }
            second: if duplicate(m) then discard(m);
                   else deposit(m, m_thread);
      }
}
```

Destination processes expecting multicast messages are connected to the RECEIVER process via FIFO delivery queues. As soon as the RECEIVER process receives a new message (*type* = *first*) from the network, say *m*, it delivers copies of *m* to the queues of the local host destination processes. The RECEIVER process also creates a new thread to monitor the progress of the multicast which gave rise to *m*. Second round messages are passed on to their respective threads.

THREAD:

```
{
      get(m);              /* get message from queue for this thread */
      start_timer(t);      /* wait for second message with a timeout */
      loop_forever
      {                                  /* start concurrent sub-threads */
            if get(m) = ok then die; /* second message, kill thread */
            if timeout then
            {
                  REL(m); /* timeout, so complete transmission */
                  die;
            }
      }
}
```

A thread picks up the first round message (passed on by the RECEIVER) and then starts a timer. After this two sub–threads are created (concurrent sub–threads within the

loop_forever statement) : one waits for the second message to arrive, after which the entire thread is killed; the other initiates a multicast to complete the protocol if the timer expires. Note that if during the multicast initiated by this sub–thread, a second round message is received by the other sub–thread, then this initiated multicast will be aborted, since the entire thread is killed. This ensures that the number of completing multicasts is limited.

With this protocol, a received message can be delivered to local destination processes soon after being received, while monitoring and completion of the multicast can be carried out concurrently. Certain optimizations to this protocol are possible to reduce the size and the number of messages. Consider the following worst case scenario: the sender crashes after completing the first round. Since there is no way a receiver can find out whether the first round completed successfully, all the receivers initiate multicasts to complete the transmission, all but one of which are unnecessary. The number of such multicasts (the majority of which are likely to be aborted half way through) however, can be reduced (to a minimum of one) by slightly staggering the timeout periods, since a thread of a RECEIVER will not initiate a multicast if it receives a second round message – which can come from some other RECEIVER's thread whose timeout expired earlier. Well known 'piggybacking' techniques can also be exploited by more sophisticated versions of the TRANSMITTER for carrying first round messages in the preceding multicast's second round messages; further the second round message need only contain the sequence number of the first round message.

4.5.1.1: Other Delivery Properties.

From the description of rel/REL_{atomic} it can be seen that in certain situations the delivery property may not be good enough to maintain consistency. For example, consider the situation in figure 4–8. Object A sends a multicast to replica group B. Only group member b1 receives the message, and A then fails. Suppose b1 produces some stable state changes (or sends a multicast message to another group) and at the same time starts to

complete the delivery protocol to ensure that all other members of B will receive the request from A i.e., b1 has determined that A has failed. However, assume that b1 fails at this point, so no functioning member of B has got the message.

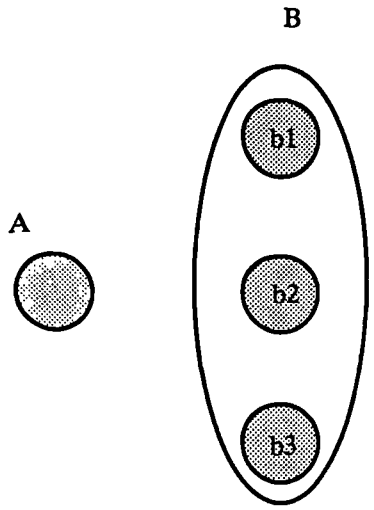


Figure 4–8: Receive Atomicity.

By the definition of rel/REL_{atomic} this is correct behaviour. However, the fact that b1 could have produced some side effects can lead to consistency problems. To avoid this it is necessary to use a communication protocol which preserves *receive atomicity*, such that the processing of any given message is delayed until all intended functioning receivers have the message. The rel/REL family of protocols includes such a communication protocol. However, in the discussions to follow, we shall use rel/REL_{atomic} for multicasting messages, assuming that the production of stable state changes take longer than the time taken to complete the delivery protocol.

4.5.1.2: Protocol Analysis and Performance.

Let t_{rel} denote the maximum measurable time duration taken for multicasting a given message to a given group by an execution of the protocol implementing *rel*. The t_{rel} is estimated to include the message queueing delays at the TRANSMITTER and RECEIVERS. Let t_d, the duration of the timeout interval used by threads created by

RECEIVER be the maximum measurable time interval within which the first round m and the second round m will be received at any functioning destination in a group, given that the sender completes multicasting m to the group. Both t_{rel} and t_d are to be estimated by considering message queueing delays at the sending and receiving nodes, message transmission delays in the communication medium and the size of the group. For the sake of simplicity, we will use the same t_{rel} and t_d for groups of different size.

The latency, L, of a protocol will be defined as the maximum time that can elapse between a sender starting a multicast to a group and a functioning process in the group receiving the message. In measuring the protocol performance, a message will be considered to have been delivered to a process when that message is deposited in the queue for that process; also, we will ignore a RECEIVER process's delay in depositing a message into the queues of the destination processes and the time taken by the RECEIVER and THREAD to execute the instructions of their respective algorithms. (This will simplify measuring protocol performance only in terms of t_{rel} and t_d).

The latency of the *rel/REL*$_{atomic}$ protocol, L_{atomic}, under a variety of situations can be expressed as:

(i) no failures: t_{rel}

(ii) sender crash after first round: t_{rel}

(iii) sender crash in the first round: $t_{rel} + t_d + t_{rel}$

(iv) worst case f crashes (sender and $(f-1)$ receivers crash during their first rounds): $t_{rel} + f(t_d + t_{rel})$.

A reasonable estimate of t_d would be to make it equal to t_{rel}, then assuming say $t_{rel} = 5$ msecs for some particular multicast, the worst case latency figure under a single crash situation (case(iii)) would be 15 msecs and the worst case figure for two crashes would be 25 msecs. Thus this simple protocol can achieve good performance despite the occurrence of failures.

Another performance parameter of interest will be the *skew* denoted as S_{atomic} and defined as the maximum time duration within which two functioning receivers are guaranteed to be delivered of a multicast message. From the analysis, S_{atomic}, can be seen to be $(t_d + t_{rel})$. This means that when a process receives a multicast message m by rel/REL_{atomic} protocol, it can assume, after S_{atomic} time, that every other functioning process in the group will have received m.

4.6: Implementation.

We have implemented a replicated RPC mechanism using rel/REL. The multicast version of the Rajdoot RPC mechanism [Panzieri 88] which this communication protocol replaces, works by multicasting a message to its destination(s) and then waiting for all of the replies, starting a timeout clock to detect failures in the remote services. If replies are not received within this timeout period then the initial call is resent. This occurs a finite number of times before the RPC mechanism assumes that the remote service has failed (similar to the V System [Cheriton 84] and the Circus System [Cooper 84b]). The multicast communication can be carried out in a number of ways. One method would be for the transmitter to send the message to each member of the group individually. Another would be to broadcast a message, relying on filter processes at nodes to discard unwanted messages. The current implementation of Rajdoot has both of these methods implemented, but it is the former method which is typically used (and was used for the following results). Note that no atomicity assurance is given: if the client fails, only a subset of the servers may be invoked.

```
rpc_call(m)
{
      start_timeout;
      retry = 0;          /* initialise retry value */
      do {
            retry = retry +1;
            send_message(m);
            receive_replies();
      } while not(timeout) and no_reply and retry != max_retry;
      if received_result then ok;
      else fail;          /* no result received in time */
}
```

Since the message to be transmitted may be larger than the maximum transmissible packet allowed by the network, *send_message* fragments the message into packets which can be sent. At the receiving side, the receiver re-assembles the message from the packets (each packet contains enough information for the receiver to be able to do this). If packets are lost then the entire message transmission is considered a failure. Since the receiver does not send acknowledgements to "corrupted" messages the sender will eventually timeout and retransmit the message.

4.6.1: rel.

Because the communications medium we use in our tests is an Ethernet LAN, the loss of packets is a very rare occurrence. As such, our initial implementation of *rel* is a simple protocol which has the ability to assemble messages even if the packets are received in the wrong order. Each packet contains enough information for the receiver to be able to reconstruct the whole message from the fragments, even if the packets are not received in the correct order. It was decided that the additional overhead imposed by having a protocol which acknowledged the reception of every packet was unnecessary in our system as packet loss is extremely rare.

```
rel(m)                          /* Transmit message */
{
      split_message(m);
      send_packets();           /* Send message packets */
}

receive(m)                      /* Receive message */
{
      receive_packets();        /* Receive message packets */
      m = assemble_message();   /* re-assemble message in correct order */
}
```

To test the implementation of *rel* an RPC mechanism was built which uses *rel*. The initial implementation is very close to the Rajdoot implementation described above. Messages are mutlicast to their destinations and then the client waits for the correct number of replies to be received. The sender will wait for a set period of time for all replies to be received. If a destination has failed then this will only become apparent when no reply is received from it, at which time the client will attempt to retransmit the message.

```
rel_call(m)
{
      start_timeout;
      retry = 0;          /* initialise retry value */
      do {
            retry = retry +1;
            rel(m);
            receive(n);
      } while not(timeout) and no_reply and retry != max_retry;
      if received_result then ok;
      else fail;          /* no result received in time */
}
```

4.6.1.1: Timings.

Table 1-1 shows the timings taken for a RPC built using just the initial implementation of *rel*. These timings were performed between SUN SPARC workstations connected by a 10 Mb/sec Ethernet. Each message transmitted was a null RPC i.e., one character in length, and used the algorithm outlined above. The figures given represent the round-trip time for a request to be sent to a server and for the reply to be received by the client.

Number	Type of Interaction	rel RPC (null RPC)
1	1 client to 1 server	7.5 ms
2	1 client to 3 servers	10.8 ms
3	1 client to 3 servers	8.3 ms
4	1 client to 3 servers with a failed server	15.12 seconds

Table 1-1: rel RPC timings.

Timing number 3 represents a client which only waited for the first reply to arrive from the replica group instead of waiting for all of the replies, as was the case in timing number 2. Timing number 4 is quoted in seconds because the client only detects the fact that a server has failed when a reply is not received. Since the timeout for this RPC was set to 5 seconds, the reply was not overdue until this time had elapsed. After that the client attempts to retransmit the message in case the original message was lost. It takes a further 10 seconds for the client to decide that the server has in fact failed.

4.6.2: rel/REL RPC.

The rel/REL implementation follows very closely the original rel/REL algorithm. An RPC mechanism was then implemented using the rel/REL implementation.

```
CLIENT

send_request(m) {
  rel(m)
  rel(m)
}

repeat
      receive_result()
      filter_result
until valid_result
```

```
SERVER
repeat
      receive_request1()
        thread_receive_request2();
        if thread_timeout
            send_request(m) { ... }
        fi
      filter_request
until valid_request
call_object_method
send_result(n) {
  rel(n)
  rel(n)
}
```

Figure 4-9: rel/REL.

The *send_request* function transmits the message *m* twice, using the *rel* protocol discussed previously. Once the sender has successfully multicast the message it can then return and call *receive_result()*. At the server side, *receive_request1()* waits for the first round of the request to arrive. It then spawns a thread to receive the second round message (*receive_request2()*), while the main thread returns to execute the request. If the second round of the request does not arrive then the thread will eventually timeout and will complete the transmission of the request (by calling *send_request*). The receiver transmits its result using *send_result*, which operates like *send_request*, in that it too uses the rel/REL protocol. The sender function *receive_result* operates in exactly the same way as the corresponding receiver function.

There are currently two implementations of the rel/REL RPC, one of which uses threads as indicated in Figure 4–9 to complete the communication protocol, and one which does not use a separate thread but instead simply waits for the second round of the protocol to be received before allowing the client or server to continue to execute. The thread based implementation, although the more efficient of the two, is currently not fully integrated with the rest of our system to allow its use for general applications and replication. As such, timings have been given for the synchronous implementation only.

It is possible to specify how many replies are required from our reliable group RPC mechanism before it returns a reply to the user. The two conditions which we use, which will be described in Chapter 6, are: *type* = *all*, meaning get replies from all functioning group members and *type* = *one*, meaning a single reply would do.

4.6.2.1: Timings.

Various timings have been carried out on the preliminary implementation of the rel/REL based RPC. These were again done on Sun Sparc machines. The timings were done using null RPCs (length of one character). Replicated clients were used to illustrate that the protocol works correctly even if one of the senders of a request fails during transmission. In the case where one of the replicated clients was faulty, it transmitted the

first round of the protocol before failing, leaving the servers to complete the protocol before replying. The other client remained operational to receive the replies from the servers. The timeout period for the protocol (the time between the first round being received and the second round being deemed overdue) was 5 ms.

Note that as has been mentioned throughout the discussion on replica group interactions, all requests and replies to replica groups are sent to all of the group members and the servers only execute a replicated request once regardless of the number of such requests which are received.

Number	Type of Interaction	State of Clients	rel/REL RPC
1	1 client to 1 server	client not faulty	11.6 ms
2	1 client to 3 servers	client not faulty	16.6 ms
3	1 client to 3 servers	client not faulty	12.2 ms
4	1 client to 3 servers with 1 server crash	client not faulty	15.28 seconds
5	2 clients to 3 servers	1 client faulty	17.1 ms
6	2 clients to 3 servers	1 client faulty	25.7 ms

Table 1–2: Non–thread based rel/REL timings.

The overhead incurred by using rel/REL occurs in the time required to receive the additional messages (approximately 2 ms per message) at both the client and server side. If threads are used then our clients/servers can resume execution as soon as the first reply/request arrives because they spawn a separate thread to receive any other messages. Thus the overhead would not be experienced directly by the client/server.

Timing number 2 represents a client waiting for all replies to arrive from the replica group, whereas timing number 3 shows a client which waited for only one reply to arrive before it continued to execute. This is why timing number 3 is approximately the same as timing number 1, as the interactions are similar.

There are two timings given for the interaction between 2 clients and 3 servers (numbered 5 and 6) because of the non–deterministic way in which messages could be

delivered to the servers from the clients. In the first timing, the servers received the request from the non–faulty client and sent a reply, which is why the time given is approximately the same as timing number 2.

The difference of approximately 8 ms between the last two results is because in timing number 6 the servers receive the first round of the protocol from the faulty client first and then timeout after not receiving the second round of rel/REL (after 5 ms) and then re–send the message (another 2 ms). Because by completing the rel/REL protocol each server transmits the request to the destinations the original client attempted to send it to, each server may again receive the request. However, it will appear as though the messages originated from the original (faulty) client, and will be considered retransmissions so the servers will not execute the request again if they received the original message. After completing the receive protocol the servers reply to the faulty–client and this reply is also sent to the functioning member of the client group.

4.7: Enhancements for Replicated Procedure Calls.

As described previously there are two sources of possible replica state divergence which cannot be solved by simply making use of only the broadcast protocols described. These sources of state divergence are local timeouts, and input message buffer overflow. What we present below are enhancements to the broadcast protocols described which provide solutions to these two sources of non–deterministic behaviour. Note that the presence of a reliable multicast protocol is assumed (rel/REL_{atomic} in this case). We shall first look at an optimization to the RPC which can result in fewer messages being exchanged between replica groups.

4.7.1: Optimizations to the RPC.

The design discussed previously can be optimized to reduce the number of message exchanges between client group C and server group S and fully exploit the capability of the

reliable multicasting service as follows: suppose that C and S become members of a composite group SG as well, as shown in Figure 4–10.

Then a call message to S sent to SG by α will be received by all other members of C; similarly if a reply message is sent to SG (rather than directly to C) then it will be received by all of 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 only if its message queue does not contain such a message. In the best case situation only one message will flow from client to server and vice versa. The scheme proposed in [Jalote 89] is similar to this.

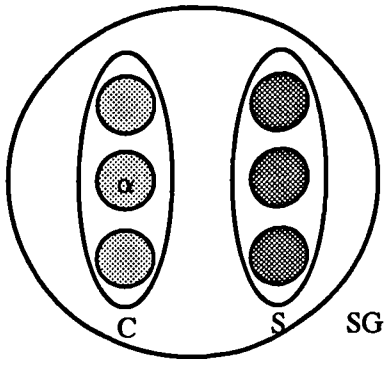


Figure 4–10: rel/REL Optimization.

4.7.2: Timeouts.

Consider the case where a client replica group (*A*, *B*, *C*, and *D*) is communicating via RPC with a server replica group. The replicated RPC requests sent to the server group are handled by a filter mechanism at each server node which identifies and discards redundant RPC requests. Similarly, a filter mechanism exists at each client node which will filter redundant replies from the server group, if the server group is made up of more than one server.

It is possible that one client node, (say *A*), experiences its RPC timeout expiring without any response from the server group being received, while another client node has

received the RPC reply before its locally set RPC timeout expired. This can happen if, for example, C (running slower than A) makes its request much later than A did, and the servers are running slower than clients assumed when they set their RPC timeout. To maintain consistency among clients in such circumstances, all functioning client nodes must agree either to regard that the RPC reply has been "timely" (even if one or more clients had their local timeout expired before the RPC reply arrived) or to discard the RPC reply as being late (even if some of the clients had received the reply before their local timeout expired).

The solution to this client consistency maintenance problem can be outlined as follows: if at least one functioning client has received the RPC reply before its local timeout expired and before it knew of any other client's expiry of local timeout, then they all agree to accept the RPC reply as timely; or, if all functioning nodes know of some other client's timeout expiry before they receive the RPC reply, then they all agree to discard the RPC reply as being late.

The consistency maintenance problem that exists at the client side can also exist at the server side when servers in a replicated server group process an RPC request with local timeouts. A fast server that received the RPC request first will be able to provide the RPC reply. However, other (slower) servers may see their local timeout expiring before they complete processing the RPC request and they will be forced to abort processing the request. Though the server inconsistency problem is less serious in the sense that it can be avoided by letting the servers assume a sufficiently large RPC timeout, the solution proposed here for the client consistency problem can be easily adopted at the server end as well.

4.7.3: The Proposed Solution.

In the description to follow, we shall assume the use of a rel/REL_{atomic}. Section 4.5.1.1 showed that in certain rare situations the delivery property offered by this protocol may be insufficient to prevent replica divergence, necessitating the use of a protocol with stronger

delivery properties. However, the use of such a communication protocol would simply impose a greater overhead on all group communications and does not affect the algorithm to be described.

If a client times out waiting on an RPC–timeout for an RPC reply, it multicasts a *timeout* message to other clients and sets up another timeout for duration D. By multicasting a *timeout* message and by waiting on the second timeout D, the client forces, and waits for, an agreement to be reached with other clients over the timeliness of the RPC request. If it receives the original RPC reply after setting the second timeout it does not process the reply but keeps the reply in its message queue until agreement is reached.

When another client whose timeout has not yet expired receives the first client's *timeout* message, it will find itself in one of two situations:

(i) it has not yet received the RPC reply.

(ii) it has already received, and possibly processed, the reply.

In the first case, the client stops the RPC timeout and sets up a new timeout for duration D. If it receives the RPC reply while waiting on the timeout D, it, like the client that initially multicast the *timeout* message, does not process the reply but keeps the reply in its message queue.

In the second case, the client multicasts an *ok* message to all other clients so that the ones that wait on timeout D (after either multicasting or receiving a *timeout* message) can agree that the RPC reply should be timely. The timeout period D is calculated so that within that period, all functioning clients will agree unanimously on the timeliness of the RPC reply, and either all or none of the functioning clients will receive the RPC reply, if the RPC reply is agreed to be timely.

4.7.4: Estimation of the timeout period.

Let $\Delta(f)$ be L_{atomic} (described in Section 4.5.1.2) in the worst case failure situation where there are f crashes (sender and (f–1) receivers crash during their first rounds). Therefore, $\Delta(f) = t_{rel} + f(t_{rel} + t_d)$.

A client that has seen its RPC–timeout expire will multicast a *timeout* message which will be received by every other client within t_{rel} time units. If there is a client that has received the RPC–reply before receiving the *timeout* message, it will multicast an *ok* message which can take at most $\Delta(f)$ time units to be delivered. Therefore, within $\Delta(f) + t_{rel}$ after the RPC–timeout expiry, the client that multicast the *timeout* message will be able to receive an *ok* message if one has been multicast at all. If that client does not receive an *ok* message within that period it will never receive an *ok* message because, either no client received the RPC reply before receiving the *timeout* message, or any client that has received the RPC reply crashed before responding to the *timeout* message, or possibly both. Let $D' = \Delta(f) + t_{rel}$. Thus, within D' time units after the RPC–timeout expired, the client that has multicast a timeout message can decide on the timeliness of the RPC.

Consider a client A that has received the *timeout* message from another client B before it received the RPC reply. Client A can assume that within t_{rel} time units the *timeout* message it received will be received by every other client if B has not crashed; otherwise, it will be within $t_d + t_{rel}$ time units, due to the thread of *rel/REL*$_{atomic}$ protocol running on its node completing the multicast of the *timeout* message. If an *ok* message has been multicast in response to the *timeout* message then that message will be received within $\Delta(f)$ time units if the first client that has initiated the *timeout* multicast (B) has not crashed, or within $\Delta(f\text{–}1)$ time units if B has crashed. Thus, following the reception of a *timeout* message, a client that has not received, but is still waiting for, the RPC reply can decide on the timeliness of the RPC no later than $\max\{t_{rel} + \Delta(f), t_d + t_{rel} + \Delta(f\text{–}1)\} = D'$ time units.

Thus, by waiting on a timeout of duration D', the client that multicast the *timeout* message or received a *timeout* message before receiving the RPC reply can reach a unanimous agreement over the timeliness of the RPC. This unanimous agreement reached is also guaranteed to be valid in the sense that if at least one (or no) functioning client has regarded the RPC reply as timely then all other functioning clients (no other functioning client) will also regard the RPC reply as timely.

Consider a client that received the RPC reply before the expiry of its local timeout and before receiving any *timeout* message from another client. This client, upon receiving a *timeout* message, will initiate multicasting an *ok* message through rel/REL_{atomic} protocol. Its host node, after receiving the RPC reply, would have set up a thread, according to rel/REL_{atomic} protocol, to monitor the completion of the server's multicast which delivered the RPC reply. This thread will wait on a timeout period of t_d and if the server's multicast is suspected to be incomplete, the thread will start multicasting the RPC reply after this timeout expires. Therefore, the maximum time difference between the client initiating the multicast of an *ok* message and initiating the multicast of the RPC reply will be at most t_d. If, however, the thread sees the server's multicast to be complete, then, by the property of the rel/REL_{atomic} protocol, all functioning clients will have received the RPC reply directly from the server. Therefore, by no later than $t_d + \Delta(x)$ time after this client has received the *timeout* message either all or none of the functioning clients are guaranteed to receive the reply, where x is the number of client crashes. Thus, by setting $D = D' + t_d$, either all or none of the functioning clients are guaranteed to receive the reply. Since $D > D'$, before the expiry of timeout D, all clients that are waiting on timeout D will reach an unanimous agreement on the timeliness of RPC timeouts.

4.7.4.1: Slow Replicas.

It is possible that replicas on slow (overloaded) nodes may receive *timeout* messages from faster replicas for servers with which they have not yet communicated. Our protocol can still function correctly because it does not depend upon replicas having processed

every message which has arrived in their message queues. The atomic nature of the communication primitives ensures that every functioning replica will receive messages, whether or not they are currently able to process them. If a replica receives a *timeout* message, it (or a separate thread which can be made responsible for handling *timeout* messages) can inspect the message queue and respond accordingly. The replica which initiated the *timeout* protocol is only interested in knowing whether any other replica has received a message from the server in question, not whether that replica has processed the message, because this means it will eventually receive the message also.

4.7.4.2: Example Figures.

Following the definitions in Section 4.5.1.2 and Section 4.7.4 we shall examine some example figures for D. If we assume that t_{rel} is 8 ms and that t_d is 5 ms, then the following table shows $\Delta(f)$ and D for different numbers of process crashes, *f*:

Number of failures	$\Delta(f)$	D
1	21 ms	34 ms
2	34 ms	47 ms
3	47 ms	60 ms

Table 1-3: Estimation of *timeout* value D.

4.7.5: Flow Control Problem.

Consider the same replicated client group consisting of *A*, *B*, *C*, and *D*. Each client has a message buffer from which it extracts the next reply message. Incoming messages which cannot be accepted directly by the client are queued here. However, since this message buffer is of a finite size it is possible that it could eventually overflow, with some messages being lost even though they were delivered to the client node correctly. This can lead to a virtual partition, where a client can appear to be partitioned from the network for some messages but not for others.

This problem can occur even if the clients were not replicated. Replicating clients and servers increases the likelihood of a virtual partition occurring. A typical way in which

buffer overflow can occur is if one of the clients is executing on a fast node (relative to the others). This client (*A*, say) sends requests and processes the replies as soon as they arrive. The replies are multicast back to the replicated client group and so the other clients will find them in their message buffers. If clients *B*, *C*, and *D* cannot deal with the inflow of messages then their buffers will eventually overflow, leading to the situation in which replicated clients could then take different actions.

4.7.6: The Proposed Solution.

The proposed solution is simply for the slower replicas, whose message buffers are about to overflow, to prevent the faster replicas from sending any further requests until the message buffers are empty (or enough space has been freed in the buffers). When a replica determines that its message buffer has grown too large then it sends a HALT message to the replica group. When a HALT message arrives in a message buffer it prevents any further request messages from being sent once it has been observed by the receiving replica. Requests and replies which have been sent but have not yet been delivered will still be received, and replicas can still deal with messages which are currently residing within the queue. If replicas have the identities of other groups with which they have been communicating, then it is possible for them to also send the HALT message to those groups, which will simply refuse to send any further requests/replies to *that* group until the HALT message has been removed. For the rest of this discussion we shall assume that only the members of the client's replica group are involved in the protocol.

When a replica transmits a HALT message it also determines how long it will require to reduce its message queue to an acceptable size, and this is transmitted in the HALT message. Receivers of the message then start a timer using this value (T_{prune}) as the timeout. If the sender of the HALT message reduces its queue before this timeout has expired then it sends a START message. START messages cancel any timers that were started because of the reception of the initial HALT. The reception and execution of a START message also implicitly removes the original HALT message from the message

queue. Replicas can then continue to transmit requests. Since it is possible for an object to be a member of more than one group it is also possible for an object to have received multiple HALT messages from those different groups. Because of this, associated with each HALT and START message is an identifier and $START_i$ (where i is the identifier) cancels only timers started with $HALT_i$.

If no START message has arrived before the timeout expires then this can mean either that the initiator of the HALT message has failed or that it is taking longer to deal with the messages in its queue than it originally expected. However, the true fate of this replica is unknown to other replicas in the group and they simply assume that it has crashed and remove the HALT message in their queues before returning to transmitting requests and receiving replies. If the replica which initiated the HALT message has not crashed and needs more time to deal with messages in its buffer then it simply starts the protocol again by transmitting another HALT message. (The object which sent the original HALT message could send another if the timeout is about to expire, to "refresh" the timeout).

Fig. 4–11 shows replicated clients *A*, *B*, *C*, and *D* with their associated message queues (only reply messages are shown here for simplicity, but if client requests are also multicast back to the client group they too would be visible). Client *A* is executing faster than *B* and *C*, which are operating at approximately the same speed but are still executing faster than *D*. The clients are assumed to be communicating with some service which is not shown in the diagram, with messages *Reply1* and *Reply2* being sent back as a result of requests (say, *Request1* and *Request2*).

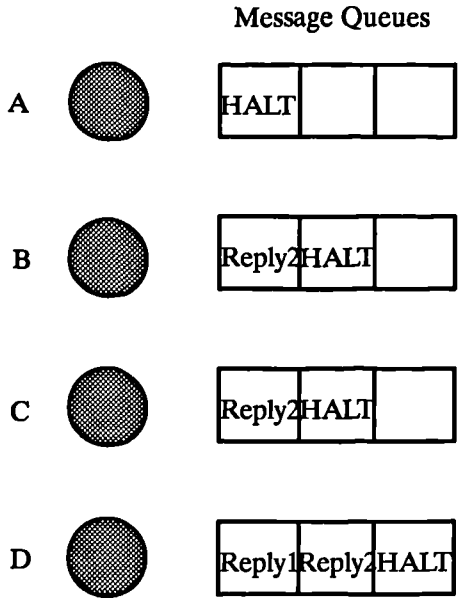


Figure 4–11: Buffer Overflow Problem.

Client *A* has sent *Request1*, processed *Reply1*, sent *Request2* and processed *Reply2* before client *D* has been able to process *Reply1*. It's message queue is about to overflow, so it starts the HALT protocol by sending a HALT message to the client group. This message appears in the message queue of all clients (including the sender). The clients then start a timer and while this is running they are prevented from sending any further requests (although they are able to process any messages which are in the queue). *D* then has the opportunity to remove messages from its queue and send a START message later.

Note that the HALT and START messages need not be placed in the client's main message queue. A separate message queue could be made available for such priority message (including the timeout message described in Section 4.7.2).

4.8: Overview of Existing Systems.

Existing distributed systems which make use of multicast communication can be categorised as using either one–to–many communication (one client interacting with a replica group) or many–to–many communication (replica groups interacting with replica groups). We shall now describe a number of the popular protocols and show how they have

approached the problems of maintaining replica consistency to both *external* and *internal* events.

4.8.1: One-to-Many Communication.

4.8.1.1: The V System.

The V System [Cheriton 84][Cheriton 85] makes use of what they term one-to-many *communication transactions*, where the start of a transaction marks the end of the previous transaction (any outstanding messages associated with the old transaction are destroyed without being processed). Each group member has a finite message buffer into which replies are queued provided they arrive before the start of the next send transaction. Although a reliable communication protocol is not used, the assumption is made that by making use of retransmissions is it possible to ensure (with a high probability) that messages are delivered if the sender and receiver remain operational. A further assumption is that at least one operational group member receives and replies to a message.

The designers of the V System recognized the possibility of message loss due to message buffer overflow, and their solution was to make use of larger buffers and to modify the kernel to introduce a random delay when replying to a group send, thereby reducing the number of messages in a queue at any time. This imposes a consistent overhead on all group replies but still does not ensure that buffer overflow will never occur (if the number of different groups interacting with each other increases then even with this delay it is still probable that the finite sized message queues will overflow).

When servers reply to a client they do so on a many-to-one basis i.e., all servers which received a request from a particular client will attempt to reply to that client (and not to any replica group the client may be a member of). This means that there is the possibility of members of the same group taking different actions as a result of local events such as timeouts (resulting in state divergence), because the servers replied to some, but not all, clients on time.

4.8.1.2: The Andrew System.

The Andrew system [Satyanarayanan 90] assumes that the communication system is reliable in much the same manner as the V System i.e., given a sufficient number of retries then any operational server can always be contacted. Communication is one–to–many and the termination conditions for a particular client–server interaction is set on a per call basis (depending on the application). The designers recognized that calls between remote processes can take an indeterminate amount of time and so if a client times out it sends an "are you alive" message to any server replica which has not responded and if a reply is returned then it continues to wait. However, because servers reply on a many–to–one basis, state divergence between replicated clients can still occur: if only one client receives a reply sent before all servers fail, then the replicated clients may take different actions.

It was recognized that messages might be lost by processes because of message buffer overflow, but in Andrew it is assumed that such loses will be detected by retries. This implicitly relies on the servers keeping the last transmitted replies to every client, but could still result in state divergence if, for example, a server group is seen to have failed by one replicated client when it retries a request, when in fact all other members of the client group received the reply before the failure.

4.8.2: Many–to–Many Communication.

4.8.2.1: The Circus System.

The Circus System [Cooper 84b] (and Multi–Functions [Banatre 86b] in Gothic [Banatre 86a]) makes use of many–to–many communication, with both client and server groups. As with the V System they assume that with a sufficient number of transmission retries it is possible to deliver messages to all operational members of a group. Thus, if a call eventually times out (after retrying a *sufficient* number of times) a sender can assume that the receiver(s) has failed. Each member of a group has a message queue which is assumed to be infinite in size (if this assumption cannot be made then the designers of

Circus assume that retransmissions of requests and replies can account for losses due to buffer overflow).

Throughout the description of Circus the assumption is made that the members of a group are deterministic (given the same set of messages in the same order and the same starting state then they will all arrive at the same next state). Client groups' members operate as though they were not replicated and do not know that they are members of a replica group. No mention is made of how this determinism of group members extends to asynchronous local events such as timeouts.

Despite the assumptions made by the designers, both the problem of message buffer overflow, and local timeouts, can occur in the Circus system, leading to replica state divergence.

4.9: Summary.

This chapter started by introducing the problems which arise in maintaining replica consistency in the presence of failures, and described how the communications subsystem can be used to given certain guarantees on the delivery of messages. It was shown how maintaining replica consistency breaks down into maintaining consistency to both external and internal events (message delivery, and local events such as timeouts). This requires the design of a reliable group RPC mechanism, which the remainder of this chapter described.

The next section described what ordering properties can be obtained from a communication subsystem (*ordered multicasts*), with a view towards maintaining consistency in a group to external events, and ended with a description of Psync, a subsystem which provides some of the ordering properties described. How such ordered multicast communication can be used in providing active replication was then shown, indicating how the State Machine conditions can be met by making use of totally ordered multicasts. However, it was noted that in many situations making use of such a delivery property can impose too much of an overhead e.g., if two messages arrive at overlapping

destinations and their respective ordering is unimportant, then using a protocol which will order them consistently is unnecessary. For this reason, we have chosen to use *unordered atomic multicasts* and to impose ordering at the application level. The algorithm and implementation of just such a multicast protocol (rel/REL) was then described, with various timings shown to indicate the overhead imposed by ensuring consistent delivery in the presence of node failures.

The next section then returned to the issue of maintaining replica consistency to internal events. It described how it is possible (given the existence of a reliable group RPC) to ensure that either all members of a replica group will determine that a local timeout has occurred on behalf of a remote service, or they will all determine that it has not occurred. This involves running an agreement protocol between the members, but since this protocol is only invoked when a timeout is observed it should not impose a great overhead on the application.

It was also shown how the number of messages exchanged between replica groups can be minimized. If client and server replica groups also receive messages sent by their own members then a given replica need not transmit a similar message if one has already been sent by another member of the same group. In the best case situation this can lead to only one message actually being sent between replica groups.

To deal with message buffer overflows another protocol was described. The replica(s) which determines that its message buffer is about to overflow can prevent the other members of its group from transmitting any further requests until it has made sufficient space in its buffer by dealing with those message which are currently there.

Finally, an overview of some RPC mechanisms was given to illustrate how they attempt to handle the consistency problems which arise in allowing replica groups. It was shown how none of them provide sufficient provision for ensuring that replica states cannot diverge as a result of either message buffer overflow, or local timeouts occurring on behalf of other remote services. None of the systems described made use of reliable

communication, instead making the assumption that given a sufficient number of retransmission of requests and replies any message will eventually be delivered if the destination is functioning. None of the systems provided a solution to the problem of consistency in the event of local timeouts, but they did attempt to solve the problem of message buffer overflows. However, none of these solutions were complete and thorough, still leading to the possibility of state divergence.

5: Object Replication in Practice.

Chapter 3 described the principles behind object replication, and outlined some of the problems encountered. This chapter will continue that discussion by examining some systems for which replication schemes have been designed. We shall use this information in Chapter 6 to show how our replication protocol was developed to operate in an integrated manner with atomic actions, relying on the underlying communications layer to impose reliable delivery of messages, and the atomic actions to impose ordering on the messages when necessary.

Although the subject of this thesis is replication in systems which use atomic actions, we shall also discuss replication in systems where actions are not used, to illustrate how those systems need to provide similar mechanisms in order that their replication protocols are consistent and correct in the presence of failures. Because most replication techniques to date have been developed for replicating data (as against objects or processes) we shall examine these protocols first and consider what changes are necessary when objects or processes are the unit of replication instead of data.

5.1: Replication and Atomic Actions.

Replication can be integrated into atomic actions in two ways: *replicas within actions,* and *replicated actions*. These two methods are substantially different from each other and yet attempt to solve the same problem: to ensure (with a high degree of probability) that an action can commit despite a finite number of component failures.

5.1.1: Replicas within Actions.

Of the two methods of combining replication with atomic actions, this method is the most intuitive: each non–replicated object is replaced by a replica group. Whenever an action issues a request on the object this is translated into a request on the entire replica group (how the requests are interpreted depends upon the replica consistency protocol). Failures of replicas within a given replica group are handled by the replication protocol in

an effort to mask them until a sufficient number of failures have occurred which makes masking impossible, and in this case the replica group fails. When an action comes to commit, the replication protocol dictates the minimum number of replicas which must be functioning in any given replica group for that group to be considered operational and allow the action to commit.

Problems which arise from this method have already been outlined in Chapter 3 and Chapter 4. The replication protocol must be able to deal with multiple messages from both replicated client groups as well as replicated server groups; the majority of replication protocols assume that replicated objects within the same replica group are deterministic; problems can occur in maintaining consistency between replicas when local asynchronous events can occur such as RPC timeouts (Section 4.7).

It is this method of combining atomic actions and replication which is used by most distributed systems, and will be examined and used within this thesis.

5.1.2: Replicated Actions.

The idea behind Replicated Actions [Ahamad 87][Ng 89] is to take an atomic action written for a non–replicated system and replicate the entire action to achieve availability. In this scheme the unit of replication is the action along with non–replicated objects used within it. In the Clouds distributed system [Ahamad 87] the replicated actions are called PETs (Parallel Execution Threads). By replicating the action N times (creating N PETs) and hence by also replicating the objects which the action uses, the probability of at least one action being able to commit successfully is increased. Although the actions and objects are replicated, each action only ever communicates with one replica from a given object–replica group and does not know about either the other replicas or the other actions until it comes to commit. Subsequently, if this replica fails, so too does the action (as is the case in a non–replicated system).

Because each action only uses one replica in a given group and each replica only receives messages from this one action, the replicas need not be deterministic, and there is

no need to devise some protocol for dealing with multiple requests (or replies) from a client (server) group. When the replicated actions commit, it is necessary to ensure that only one action does so (to maintain consistency between the replicated objects). To do this, typically the first action to commit (the fastest) as part of its commit protocol copies the states of the objects it has used to those replicas it did not use (i.e. the replicas used by the other actions) and causes the other actions to abort. This is done atomically, so that if this atomic action should fail then the next replicated action which commits will proceed as though it had finished first.

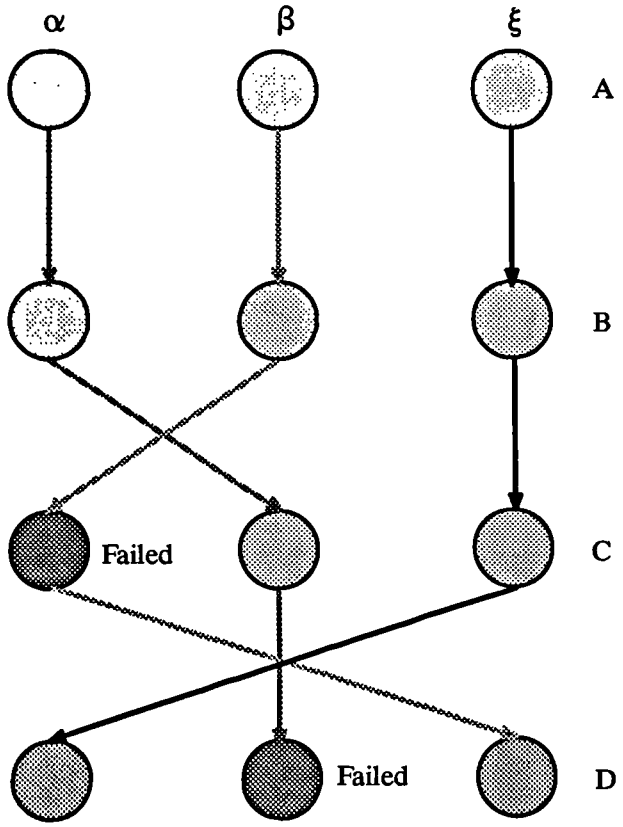


Figure 5–1: Replicated Transactions.

Figure 5–1 shows the state of the objects A, B, C, and D which are replicated three times, just as the replicated actions, which are also replicated three times, α, β, and ξ, begin commit processing. The execution paths of these actions is indicated by the lines. When actions α and β come to commit they will be unable to do so because the object D

(which α used) and C (which β used) have failed, making commit impossible. However, action ξ will be able to commit, and will then update the states of all remaining functioning replicas. Failed replicas must be updated before they can be reused.

Obviously the choice of which replicas the actions should use is important e.g. in Figure 5–1 if action ξ had used the same copy of object D as action α then the failure of this object would have caused the two actions to have failed instead of one. In some systems [Ahamad 87] the choice of which replica a particular action uses is made randomly because they make the assumption that with a sufficient number of object replicas the chances of two actions using the same copy are small. However, in [Ng 89] they propose a different approach by using information about the nature of the distributed system (reliability of nodes, probability of network partitions occurring) to come up with an optimal route for each action to take when using replicated objects. This route attempts to minimize the object overlap between replicated actions and minimize the number of different nodes that an action visits during the course of its execution.

The advantage of using replicated actions as opposed to using replicas within an action are the same advantages obtained from using a passive replication protocol as opposed to using an active replication protocol: there is no need to ensure that all copies of the same object are deterministic since the action which commits will impose its view of the state of the object on all other copies, and each replica will receive a reduced number of repeated requests from replicated clients (reduced to only one message if each action makes use of a different replica).

However, the scheme also suffers from the problems inherent with a passive replication scheme: checkpointing of state across a network can be a time consuming, expensive operation. If we assume failures to be rare (as would be the case) then this scheme will cause large numbers of actions to abort. The action which commits first will overwrite the states of the other replicas, effectively removing the knowledge that the other actions ever ran. In some applications it may prove more of an overhead to

checkpoint the state of one action in this way rather than allowing all functioning actions simply to commit.

We have now discussed two ways in which atomic actions and objects can interact in a replicated environment: replicated objects within an atomic action, and replicated atomic actions. Because the former method of using replicated objects within atomic actions is used by most distributed systems we will be using this method in the remainder of this thesis. We shall now examine various replication protocols which have been designed to manage replicated objects within such an atomic action environment.

5.2: Data Replication in Atomic Action Systems.

Data replication techniques for atomic action systems to maintain *one-copy serialisability* (*1SR*) have been extensively studied (most notably with regard to replicating databases). When designing a replication protocol it is natural to examine those protocols (and systems which use them) that already exist, to determine whether they have any relevance. This is the approach we took when designing the object replication protocol to be described in Chapter 6.

Definition: If the effect of a group of atomic actions executing on a replicated object is equivalent to running those same atomic actions on a single copy of the object then the overall execution is said to be 1SR.

Definition: A *replica consistency protocol* is one which ensures 1SR.

Because most replication protocols have been developed for use in database environments it is important to understand the differences between the way in which operations function in a database system and the way in which similar operations would function in an object-oriented environment. These differences are important as they affect the way in which the replication protocols function.

In a database system, which performs operations on data structures, a read operation is typically implemented as a "read entire data structure", and a write operation is in fact a

"read entire data structure, update state locally to the invoker, then write entire new data state back". In this way, a single write operation can also update (or re–initialise) the state of an out–of–date data structure.

In an object–oriented system, the read operation is typically implemented as "read a specific data value". Similarly, the write is "perform some operation which will modify the state of the object". The object simply exports an interface with certain operations through which it is possible to manipulate the object state. Some of these operations may update the state of the object, whilst others will simply leave it unchanged. A write operation in this case may only modify a subset of an object's state, and so cannot be guaranteed to perform an update as in a database system.

In a database system, the fact that a single write operation can update the entire state of a replica is used in replication protocols such as *Available Copies*. If these protocols are to be used in an object–oriented system then they will require explicit update protocols.

Finally, in a database system the invoker of a given operation knows whether that operation is state modifying or not i.e., it knows which type of lock will be required. However, in an object–oriented system users of a given object only see the exported interface and see nothing of the implementation, and therefore do not know whether a given operation will modify the state of the object. This difference is important as many of the replication protocols to be described implicitly assume that clients have this type of knowledge (it is used to ensure that read operations can be executed faster than write operations).

We shall now examine some of the replication protocols which have been proposed for managing replicated data.

5.2.1: Available Copies.

In the Available Copies [Bernstein 87] replication protocol, a user of a replicated service reads from one replica and writes to all available replicas. Prior to the execution of

an action, each client determines how many replicas of the service there are available, and where they are, (this information may be stored in a naming service and is accessed before each atomic action is performed). Whenever a client detects a failure of a replica it must update the naming service (*name-server*) view of the replicated object by performing a *delete* operation for the failed copy. All copies of the name-server, if it too is replicated, must be updated atomically.

When a write operation is performed all copies are written to and they must all reply to this request within a specified time (it is assumed that it is always possible to communicate with non-faulty replicas). Locks must be acquired on all of the functioning replicas before the operations can be performed, and if conflicts between clients occur then some replicas will not be locked on behalf of a client, and the client will be informed, at which point the calling action is aborted. Using this locking policy and the serialisability property of the actions within which operations occur, it is possible to ensure that all replicas execute the operations in identical order.

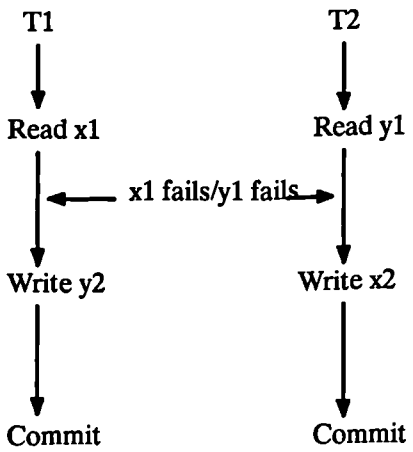
If all replicas reply to a write operation then the action may continue. However, if only a subset reply the action must ensure that the silent members have in fact failed. If the silent replicas subsequently reply then the action must abort and try again (this is because the states of the replicas may have diverged). However, if the silent copies have actually failed then the action can still commit since all available copies are in a consistent state.

Whenever a new copy is created (or recovers from a failure) it must be brought up-to-date before the name-server is informed of the recovery (before a client can make use of the replica). When this is done the copy can take requests from clients along with the other members of the group. The updating of recovered replicas can be done automatically if an out-of-date replica intercepts/receives a write request from a current transaction, as has been mentioned previously.

Consider the history of events shown in the diagram below, where T_1 and T_2 are different transactions operating on two replica groups whose members are x1,x2 and

y1,y2. Assume that T_1 and T_2 are using a "read-one copy, write-all-available copies" scheme and that there are initially two copies of objects x and y which they both wish to access. The execution of events is as shown, with time increasing down the y-axis.

If we examine the above history, it is clearly not 1SR i.e., neither the serial execution $T_1;T_2$ nor $T_2;T_1$ are consistent with the above history. Thus, the idea of "read-one copy, write-all-available copies" by itself cannot guarantee 1SR. It is necessary to execute a validation protocol before the transaction can commit to ensure correctness. In Available Copies this takes the form of ensuring that every copy that was accessed is still available at commit time, and every replica that was unavailable is still unavailable, otherwise the action must abort.



Because of the assumption made by Available Copies that all functional replicas can always be contacted, this means that this protocol cannot be used in the presence of network partitions. A node which is partitioned cannot be distinguished from a failed node until it has been reconnected. If the replication protocol assumes that all nodes which are unavailable have failed when in fact some have only been partitioned, inconsistencies can result in the replicas, as shown in Section 3.4. As such, if partitions can occur then the replication protocol must be sufficiently sophisticated that it can ensure consistent behaviour despite such failures.

5.2.2: Weighted Voting.

The Voting (or Quorum Consensus) replication protocol [Gifford 79] is a replication scheme which can operate correctly in the presence of network partitions. In this method a non–negative weight is assigned to each replica and this weight information is available to every client in the network. When a client wishes to read (write) the group it must first gain access to what is known as a *read (write) quorum*. A quorum is any set of replicas with (typically) more than half the total weight of all copies. A read quorum (N_r) and a write quorum (N_w) are defined such that $N_r + N_w > N$ (the total weight).

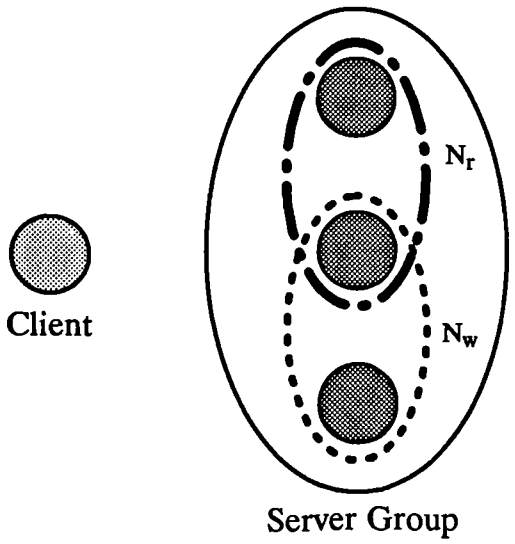


Figure 5–2: Configuration of Read/Write Quorum.

A read operation requires the access of any N_r copies (only data from up–to–date replicas should be used), and a write operation requires N_w up–to–date copies (so updates are not applied to obsolete replicas). The number of inaccessible copies tolerated by a read is $N–N_r$, and for a write operation it is $N–N_w$. The purpose of having quora is to ensure that read and write operations have at least one copy of an object in common. If the network partitions then voting allows access only from the majority partition if one exists.

Associated with each replica is a timestamp or update number which clients can use to determine which replicas are up–to–date. If $N_w \neq N$ then a read quorum is required to

obtain the most up–to–date version of this number. If $N_w = N$ then every functioning copy must contain the same value because every write operation has been performed on every replica in the group. This update number is used by clients to ensure that they only read data from up–to–date replicas, even though they may acquire access to out–of–date replicas in their read quorum.

The write operation is a two–phase, atomic operation, because either the states of N_w copies are modified or none of them are changed (to ensure that subsequent read and write quorum overlap and that a majority of the replicas are consistent). If a write quorum cannot be obtained the transaction must be aborted. However, a separate transaction can be run to copy the state of a current replica to an out–of–date replica. It is always legal to copy the contents of replicas in this way.

The weights assigned to replicas should be based on their relative importance to the system e.g., a printer spooler which resides on a very fast node would be considered better for throughput than one which resides on a slower node and would therefore be assigned a higher weight than a replica on a slower node. A replica with higher weight is more likely to be in the quorum component.

A major problem with this protocol is that read operations require a quorum, even if there is a local copy of the object. This can prove inconvenient (and slow) if an object wishes to carry out many read operations on that object in a short space of time. There is also the problem of fault tolerance: many copies of an object must be created to be able to tolerate only a few failures e.g., we require five copies just to be able to tolerate two crashes.

Note that to cut down on the storage requirements Witnesses [Pâris 86] can be used in place of actual replicas. A witness only maintains the current version number of the data. They can take part in quora, but there must be at least one "real" replica in a quorum. Other replication protocols based on the Voting protocol also exist [Abbadi 90][Jajodia 89][Davcev 89]. They address issues such as the need to acquire a quorum for a read

the responses of "true" copies. A write can only succeed if a write quorum contains at least one non-ghost copy.

Because of the way in which ghosts are created and used, a ghost is used to ensure that a particular non-ghost copy has failed due to a node crash and not to a segment partition. If it is possible to create a ghost then the segment has not been partitioned and only the node has failed. In Available Copies when a replica does not respond it is simply assumed that it is because of node failure, which can result in inconsistencies if the copy was partitioned. When the node on which the non-ghost copy originally resided is re-booted it is possible to convert the ghost copy back into its live version.

5.2.5: Regeneration.

Regeneration [Pu 86] is a similar replication scheme to Available Copies in that a client only requires one replica to service a read request but a write must be performed by all copies. When an update occurs, if fewer than the required number of replicas are available then additional copies are regenerated on other operating nodes. In doing so the system must check that there is sufficient space available on the target node for the new replica (in terms of the volatile and stable storage that it may use) and also that a copy does not already reside there. A write failure occurs if it is not possible to update the correct number of replicas, and a read failure occurs if no replica is available.

A recovering node and replica cannot simply rejoin the system. For each replicated resource the system must check to see whether the maximum number of replicas already exist. If so the recovering replica is deleted. If the maximum number does not exist the system must check one of the available replicas to determine whether the state of the recovering replica is consistent. If the recovering replica is inconsistent (i.e. an update has occurred since this replica failed) then it must be deleted because a new copy has already been created to take its place but is currently unavailable.

5.2.6: Primary Copy.

The Primary Copy [Alsberg 76] mechanism is an implementation of the passive replication strategy, where one copy of an object is designated as the primary, and the other copies become its backups. All write operations are performed on the primary copy first, which then propagates the update to the secondary copies before replying to the request. Reads can be serviced by any replica since they all contain consistent copies of the state. Any *inaccessible* secondary copies are typically marked as such (perhaps to a name-server) so that they cannot be used as either a future primary or to be read from until they have been brought up-to-date (another method would be physically to remove them from the replica group until they have been updated). If the Primary Copy fails then a reassignment takes place between the remaining copies to elect a new Primary.

A problem arises when the Primary copy fails. If the primary site is down a reassignment is in order. However, if the network has become partitioned a reassignment would compromise consistency. If network partitioning has a low probability of occurrence then the process for electing a new primary can be allowed: the secondaries should be notified of the primary's failure and they must agree amongst themselves which one is to become the new primary copy. If the election of a new primary takes place and the existing primary has not actually failed (perhaps it was not able to reply to the 'are you alive' probe-messages in time because of an overloaded node) then the protocol should ensure that the client will only accept a reply from the newly elected primary, as the old primary could be in an inconsistent state. The protocol described in [Alsberg 76] tolerates network partitions by allowing operations to continue in all partitioned segments and relying upon some "integration" protocol to merge the states of replica groups when the partitions are re-joined. However, such integration is not guaranteed to be resolvable.

If we take the case of only a fixed primary site i.e., no secondary takes over because partitioning is possible, then a resource replicated using this strategy only increases the

read availability. Its *write* availability is the same as the availability of the primary site. The other replication strategies all provide ways of increasing write availability.

5.2.7: Optimistic and Pessimistic Consistency Control.

The replication protocols described previously can all be considered *pessimistic* with regard to consistency of data in the presence of network partitions. Until a partitioned network is reconnected, it is impossible for nodes on one side of the partition to differentiate between being partitioned and a failure of the nodes on the other side of the partition. As has been described in previous sections, this can have an adverse affect on replicated groups which have also been partitioned, unless some method is provided to ensure that update operations can only be performed consistently on the *entire* group. Typically, these replication protocols are used in conjunction with atomic actions, and in the event of a network partition either only one partition (in the case of Voting) or no partition is allowed to continue to progress, meaning that any atomic actions that were executing must be aborted to maintain consistency of state between the partitioned replicas. They are pessimistic, using the principle that, if it is not possible to tell definitely that replicas have failed then it is better to do nothing at all. Those protocols which can operate correctly in the presence of a network partition (still maintain consistency of replicas), such as Voting, typically impose an overhead on the cost of performing operations on replicas (in the Voting protocol, the cost of performing a read operation is increased because a quorum of replicas must be obtained).

An *optimistic* consistency control scheme like those described in [Davidson 84][Abbadi 89] take a different approach and allow actions to continue operating even in the event of a partition. When the partition is eventually resolved it must be possible to detect any conflicts that have arisen as a result of the original failure and to be able to resolve them. These protocols assume that it is possible for committed actions to be rolled back (i.e., un-committed). How the detection and resolution of conflicts is performed is

system specific e.g., in some systems it must be done manually, whereas in [Davidson 84] a mechanism is described that will allow the system to automate much of the work.

The use of such an optimistic consistency control scheme is restricted to a subset of possible applications and systems because resolution can be time consuming, and in many atomic actions system the facility to un-commit actions is not available. This thesis will concentrate only on pessimistic consistency control.

5.2.8: Effectiveness of Replication Strategies.

In [Noe 86] a simulation study for the comparison of available copies, quorum consensus, and regeneration was carried out to determine which replication protocol was the most efficient given a specific configuration of distributed system, and a certain set of failure characteristics.

The model was programmed in SIMULA [Birwhistle 73], and assumed a local area network consisting of a number of separate computers interconnected by a communications medium such as an Ethernet, with no communications failures. The parameters used in the simulation, such as crash rates and node load, were obtained from studies of existing distributed systems and from mathematical models, and all parameters were the same for each replication protocol simulated. Crash frequency varied between 100 and 300 days, with repair times having a mean of 7 days. The number of replicated resources ranged from only one copy to having three copies, and the ratio of read requests to write requests varied from a probability of 0.3 up to 0.7, with request frequencies varying from between 50 and 400 requests per day. The number of nodes in the system also varied from 10 to 30. All measured results were taken over a simulated time of 2 years of operation.

5.2.8.1: Simulation Results.

The quantities calculated from the results were the read and write availability of the replicated service. The read availability was defined and calculated as the total number of

successful read requests divided by the total number of read requests. Write availability was similarly defined in terms of write requests.

What was found from the results was that replication provides a significant increase in availability. However, there is little point in going beyond a maximum of two copies. Both the Available Copies and Regeneration techniques provide a substantial increase in availability, raising the value of read and write availability very close to 1.0 i.e., whenever a request is performed upon a replicated resource it will be carried out successfully. There is very little additional gain with either of these protocols in having a maximum of 3 copies of each resource.

The Voting protocol provided less protection than either of the other protocols and would not even be considered until a maximum of 3 copies were used. In such a case the optimal size for a read and write quorum is 2; with a write quorum of 3 the replicated resource performed worse than in the non–replicated case because there are three ways to lose a single copy and destroy the write quorum.

Both Available Copies and Regeneration are preferable to Voting if network partitions are rare, or if measures are added to prevent or reconcile independent updates during partition rejoining. The read and write availability of the Available Copies technique are the same, and remain relatively constant despite changes in the request rate and the number of nodes. Regeneration can be preferable to Available Copies in an unstable environment that suffers from high crash frequencies, with a high number of updates and frequent reconfiguration of the network. Further, Regeneration can equal or surpass the performance of the Available Copies technique only if enough additional storage is supplied to allow regeneration, as was described in Section 5.2.5.

5.3: Object and Process Replication.

We shall now examine replication protocols which have been designed to operate on objects and processes, as opposed to pure data. As we have already seen, object or process

replication is more difficult than data replication as objects and processes can be active and issue further requests on other replica groups.

5.3.1: Coordinator-Cohort Replication.

ISIS [Birman 88][Birman 87a][Birman 87b] is a distributed system which provides a number of communication protocols which possess different delivery properties. ISIS addresses the problem we described in Chapter 4 of using cheaper communication protocols (weaker delivery properties) for certain applications. In ISIS it is possible to use a protocol which reliably delivers messages but guarantees that only those messages which need to be received in an identical order by all destinations will be e.g., an electronic mail service can ensure that all messages from a given user are delivered in the order that they were sent, but mail messages from different users may be ordered differently at overlapping destinations. This allows applications to use cheaper delivery mechanisms where they are required, using the syntactic knowledge of the application and its interactions, but the system still ensures that state changes occur in a consistent manner.

There are three broadcast primitives available in ISIS, each providing different functionality: ABCAST, CBCAST, and GBCAST. All of the primitives are *atomic* i.e., either all *operational* destinations of the message receive it or none do. There is also an implicit assumption in ISIS that network partitions do not occur. If this were not the case then the communication protocols could not guarantee delivery of messages to all operational processes.

5.3.1.1: ABCAST Communication.

ABCAST primitive: this primitive delivers messages atomically and in the same order everywhere i.e., it is an atomic multicast as described in Section 4.1.1.3. If two concurrent messages are sent by different client processes using ABCAST then a delivery ordering will be picked by the system and preserved at all overlapping destinations. ABCAST uses a multi-phase commit protocol similar to those described in Section 4.2.

5.3.1.2: CBCAST Communication.

CBCAST primitive: this communication primitive preserves only causal relationships between events. A causal communication system will only preserve an ordering of events if the order is causally related. If two events are not related in this way then there is no guarantee on the delivery order. The functionality provided by CBCAST is the same as that provided by the *Psync* system described in Section 4.3 and it is implemented in a similar way by making use of a history of messages which is transmitted along with every message.

5.3.1.3: GBCAST Communication.

It is entirely possible to construct an application using a mixture of both ABCAST and CBCAST primitives. It should be noted however, that ABCAST and CBCAST are unordered with respect to each other. GBCAST however, provides totally ordered broadcasts (Section 4.1.1.5) which are ordered with respect to every other type of broadcast i.e., if X is a GBCAST and Y is some other broadcast type then X and Y will be ordered consistently at every overlapping destination. When a process failure is detected all operational processes are informed using a GBCAST and this will ensure that they will record the failure of the process *after* they have received all outstanding messages from that process.

5.3.1.4: Replication in ISIS.

The Coordinator–Cohort replication mechanism [Birman 85][Birman 87b][Birman 88] which is used in ISIS is related closely to the Passive Replication protocol described in Section 3.3.2. Consider figure 5–3 in which there are three copies of the same replicated resource which form a single replica group.

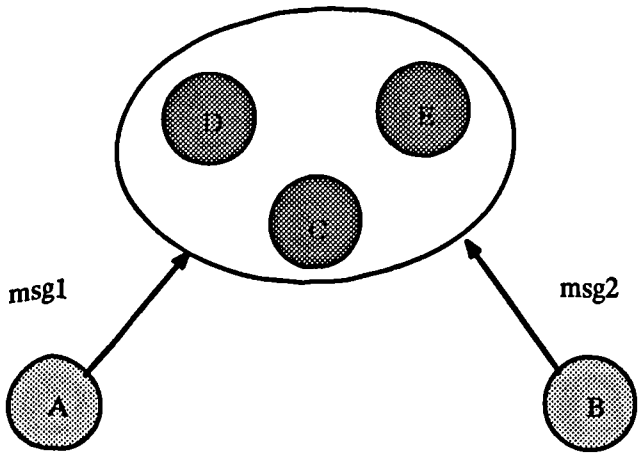


Figure 5–3: Coordinator–Cohort Replication.

In ISIS each member of a group maintains information about the other functioning members of the group. This information is kept up–to–date by a failure detector in the system which monitors all objects, and whenever a failure of an object is detected it informs all of the remaining objects using GBCAST so that the failure is ordered consistently.

In figure 5–3 there are two clients (*A* and *B*) which require access to this group. Suppose that *A* and *B* concurrently send requests to the group for a service. Such group requests are sent using the ABCAST primitive, so that all functioning replicas receive them in an identical order. Instead of both clients interacting with the same primary replica as is the case in Primary Copy, in this replication scheme each client will be serviced by a different member of the group (which becomes the client's coordinator), with the remaining members acting as the backups (the coordinator's cohorts). Because each replica in the group has an up–to–date record of which other members are in the group, by using the same deterministic algorithm all replicas in the group can unilaterally reach the same decision about which replica will become coordinator for a given client. Thus, in this diagram *C* may be chosen by all of the replicas to service *A*, while *E* may be chosen to service *B*. A given replica may be coordinator for several requests, whilst at the same time acting as cohort for another coordinator.

5.3.1.5: Checkpointing of State.

Once a request has been completed a checkpoint/update is made to the cohorts so that they can update their own states, and the client is informed. The checkpoint and reply are typically sent as one message using CBCAST. It is therefore guaranteed that all functioning group members will have received the update message and updated their states before any future requests are executed which may depend upon the initial request and reply. Causality also ensures that if there are multiple coordinators at any given time also transmitting checkpoint messages then only those messages which need to be ordered at each replica will be ordered, thus improving the performance of the replica group. If ABCAST was used to transmit the update messages then every replica would have a consistent order imposed on the checkpoint/update messages even if such message are unrelated i.e., can be delivered in an arbitrary order.

5.3.1.6: Concurrency Control.

If the operations received by multiple coordinators require exclusive access to the replicated resource then the coordinators must lock the group in the appropriate manner, by broadcasting a lock request to the group members. ISIS ensures that only one coordinator will be able to acquire the lock.

5.3.1.7: Failure Detection.

In ISIS, every node maintains a Site Monitoring process which monitors the current state of every process on the node i.e., whether they are operational or not. If a process is detected to have failed then the Site Monitor sends a *failure GBCAST* message to all processes in the system which ensures that the failure is observed by all processes consistently. When a node recovers this information is again propagated to other nodes using *recover GBCAST*. This ensures that the recovery of a node is observed after its failure (if the order was arbitrary then inconsistencies could arise).

5.3.1.8: Coordinator Election.

The election of coordinators is carried out with no exchange of messages between the members of the group. This is because each group member maintains a list of exactly which replicas are currently members of the group. This group information is formed when the group is initially created, and events such as group member failures and recoveries are observed by all functioning replicas in a consistent manner (as these messages are transmitted using GBCAST). As such, each functioning group member will have the same group membership information when the election of the coordinator takes place. By making use of this group view information and the same deterministic rule, all members come to the same decision about which replica should service a particular client request. The deterministic rule used is based on the physical location of the client (a replica on the same node as the client will always be preferred) and load balancing information.

Once a particular replica has been assigned to a client as its coordinator, it will begin to interpret the request; all of the other members will remain passive as far as this client is concerned (become cohorts) but will await any further requests from other clients. If the original coordinator fails then the Site Monitor will inform the cohorts and a new coordinator is elected which will continue the operation from the last checkpoint it received from the now failed coordinator. This process of electing new coordinators to take over from failed coordinators continues until either the request has successfully terminated or no further group members are available.

If a coordinator fails before replying, exactly one cohort will take over. If the coordinator replies before failing and the reply is delivered, then no cohort will take over because the request has been completed successfully. Thus, the coordinator–cohort scheme can guarantee a form of atomicity of execution: as long as the entire replica group does not fail, an operation will be performed and a reply will be sent. If the group does fail

then the caller will be informed but will be unable to determine at which point the execution failed.

5.3.2: Lazy Replication.

Lazy replication [Ladin 90] and other similar protocols [Downing 90][Garcia-Molina 90][Alonso 90] do not insist that all operations within an application are performed in the same order everywhere, and are hence termed *weakly consistent* replication protocols. They allow application programmers to exploit syntactic knowledge of the application to use cheaper, more efficient protocols where possible whilst at the same time maintaining the 1SR property for the application's (distributed) state.

A *weak-consistency* method represents a trade-off between consistency and availability. It allows temporary inconsistencies to develop among the replicas as a result of multiple users independently updating different replicas and the time necessary for an update to propagate to all replicas. The system guarantees to resolve these inconsistencies using techniques based on syntactic dependencies among conflicting actions, the global order in which actions were executed, or the semantic properties of operations and applications, and return the replicas to mutual consistency.

5.3.2.1: The Ordering Protocols.

Lazy replication is designed to run in the Argus [Liskov 87a][Liskov 88] distributed system. Argus uses objects and actions, where operations on objects are performed within atomic actions. Such objects may be replicated and the standard replication scheme provided ensures 1SR. However, to allow for weaker consistency ordering users can make use of the three communications primitives provided by Argus which allow the order of operations to be specified on a per operation basis:

- client-ordered: this allows clients to define the order of calls to service operations either implicitly or explicitly. The system guarantees that the service will perform operations consistent with this order. The order resulting from the implicit

specification corresponds to the causal order, while the explicit specification allows the definition of weaker orderings.

- server–ordered: this is an atomic communication protocol with the system guaranteeing that all server–ordered operations will be performed in the same order at all destinations. If concurrent server–ordered operations occur then an order will be chosen non–deterministically and followed by all functioning destinations i.e., if multiple server–ordered operations are received from different sources at overlapping destinations then some order will be chosen and followed by all of the destinations.

- globally–ordered: this is a totally ordered communication protocol which is used where a globally ordered message delivery is required.

Lazy replication is primarily concerned with client–ordered interactions, where operations can be ordered arbitrarily at different replicas, provided that such orderings maintain consistency between the replicas. Server–ordered interactions ensure that all replicas receive messages in the same order, and will be described in Section 5.3.3. The principle behind lazy replication is that performance improvements can be obtained for an application by exploiting the syntactic knowledge of the application and enforcing strict ordering only where necessary. In lazy replication, a client performs operations on a single replica, but the system guarantees that the replicated service will remain consistent i.e., that all replicas within a group will *eventually* have the same state. Effects of calls which are performed on a single replica are propagated to other replicas by a *lazy* exchange of *gossip* messages.

5.3.2.2: System Assumptions.

Before proceeding to a description of how lazy replication works, it is important to review the system assumptions made by the designers. A *service* is implemented by a fixed number of replicas (objects) which reside at well known locations. These replicas work as an active replica group, with the state of a replica kept in stable storage. A service provides

two kinds of operations: *update* operations that modify the state of the replica, and *query* operations that observe the state.

Nodes may fail, but always recover in a finite amount of time. All operations are individually atomic (multi-operation actions are not supported) so that if an update operations completes then the changes it has made are stored on stable storage, along with sufficient information to identify the message which caused the state transformation.

5.3.2.3: Client-Ordered Replica Groups.

Clients indicate the ordering between operations by transmitting ordering information with each call. This information indicates which other operations must have been performed before this operation can be executed. Since the service respects only those dependencies explicitly defined, the client must ensure that all intended ordering requirements are conveyed to the service. This can be done automatically by the system, resulting in a causal ordering, but greater performance benefits can be obtained if the client does this explicitly.

When a replica receives an update operation it can be processed immediately, even if some other messages upon which this message may depend have not been received. To do this, the actual operation is registered but not actually performed until all undelivered messages have been received. Query operations must be postponed if messages upon which they depend have not been received. This allows for faster write operations at the expense of possible delays for read requests.

Every client runs a *front end* at its node. When a client invokes an operation on a service this front end sends an appropriate message to one of the replicas in the service group, which executes the request and sends back a reply. The front end may send the request to several replicas if the initial response is slow in arriving, or if the client wants a response from the fastest replica available. Each update request transmitted is assigned a unique identifier which is used to ensure that such operations are performed only once at each replica. The replicas in a system supporting lazy replication are therefore active in

that they can all potentially receive and process the same request from a client (as in active replication) but typically only a single member of a replica group will receive the request and respond to the client and then inform the other members of the group (as in passive replication).

All replicas maintain a log of update operations they have received and periodically they construct *gossip* messages from this and transmit the information to the other replicas. This information contains the actual messages received as gossip messages are the usual way that replicas learn about state changes which have occurred at other replicas within their group. When a gossip message is received each replica will remove from it the update messages it has not yet received and order them with respect to other messages it has received. Since each replica receives a copy of the gossip message it is possible that a replica could receive multiple copies of the same request (if it received the initial update request as well as a subsequent gossip message which also contained the same request). However, because each update message is associated with a unique identifier it is possible to ensure that a given replica only executes each update request once.

5.3.2.4: Missing Updates.

It is possible that a replica which receives the only copy of an update operation fails after successfully replying to the sender but before transmitting a gossip message. If the same sender then issues a subsequent query operation this operation will go to another replica. However, because no other replica in the service group has received the original update message this query operation must be suspended. The designers point out that this situation can be made unlikely by sending frequent gossip messages before or in parallel with the update response. Another method would be for an updated replica to notify some number of other replicas (backups), and await acknowledgments before sending the user a response.

5.3.3: Viewstamped Replication.

The Viewstamped replication scheme [Oki 88] employed in Argus [Liskov 87a][Liskov 87b][Liskov 88] is based on the passive replication technique. The object and action paradigm described in Section 2.2.4 is used in Argus. Argus objects are called *guardians*. The logical unit of replication within Argus is the guardian, and replication is achieved by creating *guardian groups*, which consist of several instances called *cohorts*, which behave as a single logical entity. The cohorts can be accessed on an individual or group basis. The set of cohorts is the group's *configuration*. Each cohort knows the group and the configuration to which it belongs.

A distinguished cohort is designated as the *primary*, which will execute handler calls and take part in the two–phase commit. The remaining cohorts are called backups. The membership of a guardian group is called a *view*. If the primary fails then a backup will be elected to take over. Such reorganization is termed a *view change*, and the algorithm that carries it out, the *view management algorithm*. View changes are necessary whenever a cohort fails (whether it is a backup or a primary).

5.3.3.1: The View Management Algorithm.

To preserve the abstraction of a single–object image it is necessary to mask the failures of members of the replica group automatically and efficiently. In Figure 5–4, the view that client F, and the members of G have of the group is *v1* = {a: b c d e}, where a is the primary, and b, c, d, and e are its backups, and *v1* is the *viewid*. (The configuration is {a b c d e}, and the view is always a subset of the configuration). Note that what is not shown in the diagram is the interaction of the backups with the primary. Such interaction is necessary for the backups to monitor the current state of the primary i.e., whether it is operational or not.

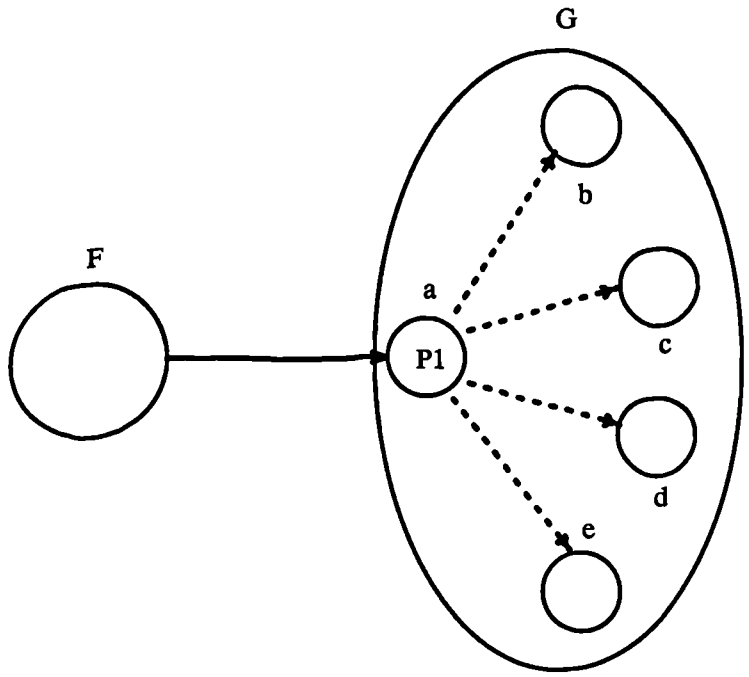


Figure 5–4: Guardian Group.

Figure 5–5 shows the same group but with a failure in the initial primary (this could be a node failure). The remaining cohorts find that they can no longer communicate with the original primary and so a new primary must be elected. A new primary is chosen from the remaining backups as long as the backups and new primary constitute at least a majority of all cohorts in the guardian configuration. In the case of the group shown, this condition has been met and so b, say, can be elected as the new primary.

In this case, the new view for F would be *v2* = {b: c d e}. If cohort a is actually operational but cannot communicate with its backups because of communication failures, it too will attempt to execute the view change algorithm. However, because it cannot communicate with a majority of the cohorts in G this view change will be unsuccessful. Cohort a will remain in view *v1*, but will become *inactive*, which means that it will refuse to execute requests on behalf of clients.

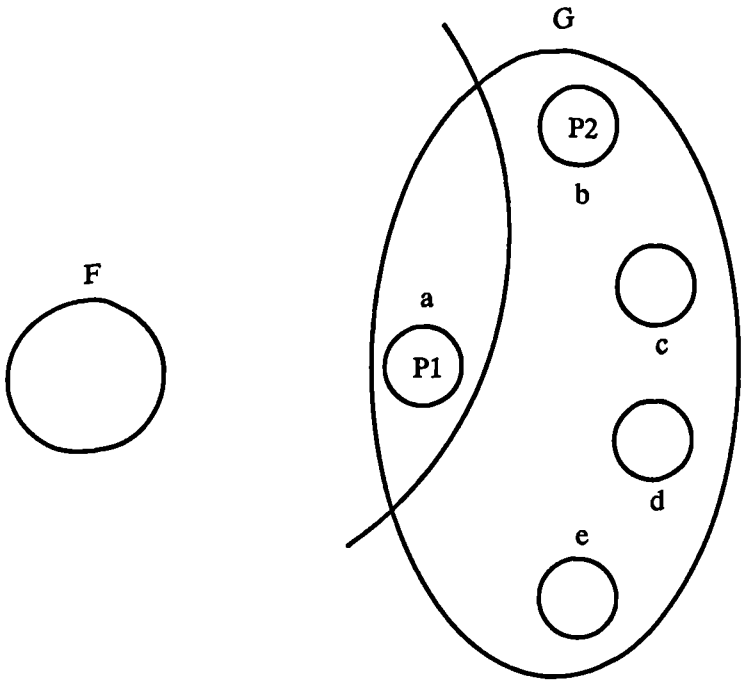


Figure 5-5: Guardian Group view change.

When a view change occurs all cohorts in the new view will be initialised with a new state. This state is the most recent state that any cohort in the group has (and this cohort will be elected as the new primary). It is guaranteed that at least one cohort in the new view will have the most recent state that the old primary sent as a checkpoint. This can be guaranteed because the old view must have contained a majority of cohorts, and the current view consists of a majority, so they must have at least one cohort in common that was in both views. Therefore the new view "knows" what happened in the previous view, and logically it must also know about everything that happened since the creation of this guardian group.

5.3.3.2: Primary Election.

Cohorts in the group periodically sent "are you alive" messages to each other. If a failure is detected then a view change is necessary and the cohort which detected the failure will attempt to contact a sub-majority of the other cohorts and get them to agree to change the view. If an agreement is reached then the view changes and the cohort with the

most recent state becomes the new primary (this is done using *viewstamps* to be described below).

5.3.3.3: Atomic Action Processing.

When a view change occurs in a guardian group, it is necessary for a user of the group to be able to tell whether the new primary "knows enough" to allow the actions which interacted with the group prior to the failure to commit. The viewstamped replication scheme uses *timestamps* and views to determine what is known by a given primary.

When a primary communicates with its backups (termed an *event*) it generates a new timestamp which is guaranteed to be unique within a view, and which form a total order. An *event record* identifies the type of the event, and contains the event's timestamp and other relevant information. Each event is assigned a timestamp, and primaries send event records to the backups in timestamp order. Each backup receives the event records in timestamp order, and must execute them in this order. Therefore, if a backup knows about event x it must know about the events which preceded x.

A *viewstamp* is a tuple consisting of a timestamp and the view in which it was generated. Each cohort maintains a *viewstamp history* that represents the sequence of events from all views seen by the cohort. For example, a viewstamp history for a given cohort might be as follows: $<v1, 5>, <v2, 10>$. This means that the state of this cohort reflects all events that occurred in view *v1* up to timestamp 5. Then the view changed to *v2* and the cohort received all events up to timestamp 10, which is the current view and the most recent timestamp it received from the current primary.

When a client makes a handler call to servers it includes the viewid in the call along with other relevant information. The primary of the group processes the call and assigns the call a new viewstamp. This viewstamp is returned to the client which remembers it in association with the group identifier. When the top-level action comes to commit, the client has a collection of viewstamps (there is at least one viewstamp associated with each server which participated in the action). The client acts as the coordinator for the action

commit protocol and sends *prepare* messages to the primaries; each prepare message contains the relevant viewstamp, which represents what *must be known* by the primary in order for the action to be able to commit.

When servers receive prepare messages they can compare the viewstamp contained within, with the viewstamp that they currently possess (which represents what *is* known). By checking that the received viewstamps are less than, or equal to the associated viewstamp in the viewstamp history, it is possible to determine whether the servers remember enough to commit the top–level action.

5.3.3.4: Example Interaction.

In Figure 5–6 client F is the coordinator for a top–level action involving guardian group G. When the group was created the initial view that F had of the group was $v1$ = {a: b c d}. This primary *a* received and processed correctly 8 operations during this view before a view change was initiated because *d* was detected to have failed. This then resulted in the new view $v2$ = {a: b c} being created. Once the view change had been completed F continued to send requests to the group, and at the time of Figure 5–6 has sent 6 requests (the primary has seen 10 events during this new view from various clients, and has checkpointed 9 of these to its backups).

F now sends the prepare message to G along with the viewstamp history that it requires the primary to know about. The primary knows about everything in view $v1$ that is required, and knows more about view $v2$ than the coordinator requires (other events have occurred which do not have any relevance for this top–level action). In this case, cohort *a* knows enough for F to be able to commit.

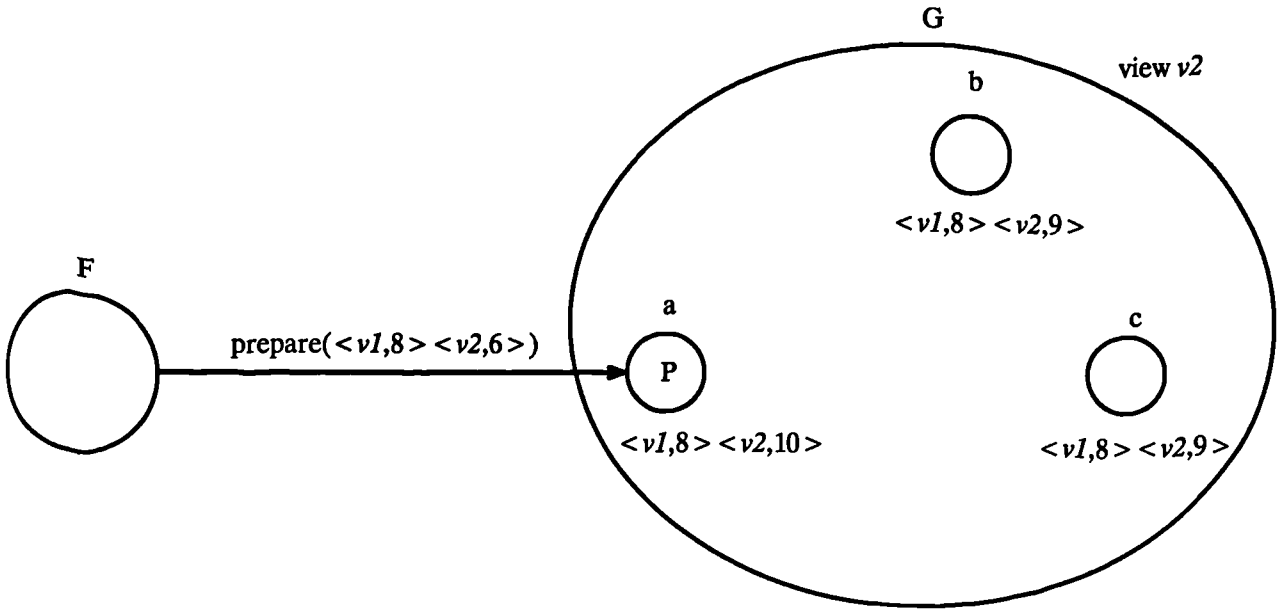


Figure 5–6: Committing a top–level action.

Even though the primary knows enough for the action to commit, it is necessary to ensure that the events are known at enough backups to survive a view change. The primary ensures this by forcing event records to the backups in the current view and waiting until a *sub–majority* know before responding *ok* to the coordinator. (A sub–majority is one less than a simple majority of the configuration).

5.3.4: Delta–4.

The Delta–4 System [Barrett 90] is specifically designed to address the requirements of real–time systems with respect to both throughput and response. The system supports the real–time concepts of priorities and deadlines, and reflects these in its communications protocols, which implement reliable group–to–group communication. As in ISIS and Lazy replication, the communications layer provides a range of primitives, each with different ordering and delivery characteristics.

In Delta–4 replicated software components are used to provide fault–tolerance. As was shown in Section 3.4, different replication protocols can tolerate a different range of failure classifications. As such, Delta–4 provides a number of replication protocols for

use, depending upon the failure characteristics of the application and the distributed system's components.

If the system is to tolerate arbitrary failures than an active replication protocol using voting, is provided which uses the states of the various replicas to detect any such failures and mask them as long as a majority of the replicas agree. If fail–silent nodes are being used in the system then voting is not necessary since messages generated by replicas can be assumed to be correct. However, as has been indicated in Section 3.3.4, active replication requires some form of atomic multicasting to ensure that all functioning replicas receive the same set of messages in the same order (to satisfy State Machine conditions C1 and C2). Such protocols are necessarily complex. It is important that such replicas are deterministic as they are required to respond quickly to external events through some form of pre–emption as is typical in dynamic real–time environments. Therefore the difficulty in maintaining consistency is compounded. Pre–emption is complex and costly to synchronise between active replicates since each replicate must be preempted at exactly the same point in its processing. The replication schemes discussed before do not support pre–emption. The *leader/follower* scheme of Delta–4 to be described next is capable of dealing with pre–emption.

5.3.4.1: Leader/Follower.

The leader/follower model of replication is a combination of active and passive replication techniques, designed for use on fail–silent nodes. All replicas are active, receiving and executing requests from clients, but only one replica (the leader) propagates output messages. The leader is responsible for taking all decisions which affect replicate determinism; such decisions are propagated from leader to followers via synchronization messages. System nodes are assumed to be fail–silent; messages are sent by the leader to the followers immediately they are generated, and when the followers generate the same messages they will be automatically discarded by the communication system.

5.3.4.2: Synchronization.

In leader/follower, State Machine conditions C1 and C2 are met as follows: atomic (unordered) multicast is used to ensure C1; C2 is ensured by imposing the choice of the leader onto its followers. Therefore, when the leader selects an input message it also constructs a synchronization message containing the identity of that input message and sends this to its followers. When the followers receive the synchronisation message they use it to execute the original input message in the same order as the leader, thus preserving replica determinism.

Leader/follower provides a solution to preserving replica determinism in the case of pre-emption by making use of the concept of *pre-emption points*; these are predefined points in the processing of an object at which it may be pre-empted. To achieve pre-emption synchronisation, each time the leader reaches a pre-emption point it increments a counter. When a message arrives at the leader, a check is made to determine whether this message requires the leader to be pre-empted. If so, the pre-emption point at which this will occur is selected (the current counter plus 1 represents the next pre-emption point) and assigned, and a synchronisation message is sent to the followers. The followers will then use this message to process the pre-emption at the correct point (this obviously requires the followers to be executing at least one step behind the leader, where a step represents the receipt of a synchronisation message due either to a pre-emption, or to the consumption of an input message by the leader).

5.3.4.3: Leader Election.

The leader is elected by a static ranking of the replicas which is created when the service begins. Subsequent leaders are chosen from this ranking. Leader failures (and follower failures) can be detected in three ways:

- failures of replicas can be detected by the communication system when it attempts to deliver a message to the destinations.

- the Delta–4 system has the notion of a Group Manager which maintains its own view of the state of the system: which nodes are currently active, which objects currently exist within the system, and which groups they are associated with, and the identity and status of each replica of an object.

- leaders periodically send "I am alive" messages to the followers.

When a leader is detected to have failed the followers will be informed and the next ranked replica takes over. There is no need for an exchange of messages between the followers to elect the new leader.

To summarize: the leader/follower approach has two advantages over the schemes presented previously:

- the leader can begin processing a message as soon as it arrives (there is no need to wait for an agreement on order to be achieved).

- the leader/follower scheme includes the provision for pre–emption.

5.3.5: Psync Replication Protocol.

The *Psync* communication subsystem was described in Section 4.3.1. We shall now describe how applications can make use of the delivery properties guaranteed by *Psync* to allow active replica groups to be used in a consistent manner despite failures.

Because active replication is used it is important to enforce a consistent ordering of operations on all members of the same replica group. A total ordering of operations is possible, but as shown in Section 5.3.2 this can be overly restrictive if the actual order in which operations are executed does not matter (e.g., two read operations could be ordered differently by the same replicas). The replication scheme used in *Psync* takes advantage of both the partial ordering of messages preserved by *Psync* and the semantics of the operations provided by the service to be replicated i.e., whether they are commutative or not.

The *Psync* approach is to first order the operations of an application based on the partial ordering e.g., operation i_1 is executed before operation i_2 because it was invoked first, and then to take advantage of the commutativity of the operations to enhance concurrency. Operations which are not commutative must be executed in the same order by each application manager and they must be totally ordered with respect to the commutative operations i.e., a *precedence* is assigned to the operations and this is used to break ties between operations that were invoked at the same logical time.

If we initially assume that none of the operations provided by an application are commutative then they will have to be sorted into the same total order at each host where a replica resides. In *Psync* this can be done by having each manager do the same topological sort of the context graph. Before the sort can be performed, each process must wait for a portion of its graph to become stable to ensure that no future messages sent to the conversation will invalidate the total ordering. *Psync* defines a *wave* to be a maximal set of messages for which the context relation does not hold between any pair. A wave is *complete* when it can be guaranteed that no further messages can arrive which belong to this wave. The topological sort proceeds from one wave to the next as the waves complete. When a message is stable all future messages must follow it in the context graph, therefore a single stable message in a wave implies that all possible members of the wave are contained in the participant's view i.e., as soon as a single message in a wave is stable, the wave can be sorted according to some deterministic sort algorithm and this sorted order gives the total order of the messages.

If some of the application's operations are commutative then the ordering can be slightly different at each conversation member because two or more commutative operations can be executed in any order. In particular, different managers can order commutative operations differently as long as there are no non–commutative operations between them. More formally, an *op–group O* is defined to be a set of operations in the context graph such that:

1: *O* contains exactly one non–commutative operation, or

2: *O* is a maximal set of commutative operations, such that for any path in the context graph between any two operations in *O*, there are no non–commutative operations.

If there exists some op–group α that contains a non–commutative operation and some other op–group β that contains a set of commutative operations, then one of the following two cases must be true: either the non–commutative operation in α precedes zero or more of the operations in β and is at the same logical time as the remaining operations in β, or the non–commutative operation in α is at the same logical time as one of more of the operations in β and is after the remaining operations in β. While managers must generate the same total order of op–groups, they may invoke the operations within each op–group in an arbitrary order.

5.3.5.1: Generalized Algorithm.

The generalized algorithm that all participant managers within *Psync* use to determine when to execute replicated operations can be outlined as follows. Assume a system of replicated objects where there are n different operations that can be applied to these objects. Assume that these operations can be subdivided into k disjoint sets $S_1, S_2, ..., S_k$ where any two operations within a set are commutative and any two operations from different sets are not commutative. Also, let different invocations of an operation in sets $S_1, S_2, ..., S_j$ $(j \leq k)$ be commutative. For such a system it is possible to define the properties $A_2, A_3, ..., A_k$:

Property Ai: All the leaves in the view are the unexecuted operations from the sets $S_i, S_{i+1}, ..., S_k$ or they are in the context of some unexecuted operations from the sets $S_i, S_{i+1}, ..., S_k$.

Observer that operations in set S_1 can be executed immediately if they are not in the context of some unexecuted operations from the other sets. In order to execute operations from S_2, property A_2 must be satisfied and the wave containing the S_2 operation must be

complete i.e., to execute operations in S_2, a total ordering is required. Since a total ordering is required it is possible to execute all the operations from sets $S_2,...,S_k$ in the current wave as soon as the property A_2 is satisfied and the current wave is complete. Thus, in the case where object operations can be subdivided into more than two sets of commutative operations, the partial ordering is used to execute the operations in the first set and total ordering is used to execute operations in the other sets i.e., *Psync* only allows additional concurrency for the first set S_1.

5.4: Summary.

This chapter started with a description of the two ways in which atomic actions and replication are typically integrated: replica groups within an atomic action, or replicated atomic actions. The various advantages and disadvantages were discussed, and it was shown that replica groups within atomic actions is the more flexible way to integrate the two mechanisms. It is the scheme that we shall be using in the next chapter.

We then moved on to a description of replication protocols proposed for use in 'conventional' systems which also use atomic actions. It was shown how most of these systems use data as their unit of replication, and hence the problems involved in replication are less complex. The replication protocols described were available copies, voting, regeneration and primary copy, and it was shown how they can tolerate failures and still continue to provide a consistent service.

The next sections described the replication techniques used in distributed systems whose units of replication are not data, but objects or processes. The systems studied included ISIS and Argus, which introduced the notion of lazy replication techniques i.e., where the relative order of message delivery is unimportant it is possible to use cheaper delivery protocols rather than the more expensive total order protocols typically used to ensure replica consistency. In the next chapter we shall see how such principles have been used in the design of a replication protocol for use in the Arjuna distributed system.

6: Replicated Objects in Arjuna.

This chapter describes some replication schemes which have been designed for the Arjuna distributed system. The first scheme has also been implemented, and makes use of the replicated RPC whose design and implementation was discussed in Chapter 4.

6.1: Arjuna System Overview.

The Arjuna system [Shrivastava 91][Dixon 89] provides tools to assist the construction of fault-tolerant applications structured as atomic actions operating on persistent objects. The Arjuna system is implemented in C++ and extensively uses the type-inheritance facilities provided by the language. User objects can inherit desirable characteristics such as persistence and recoverability. We shall be using inheritance and the *stub generator* [Parrington 89] to create transparently replicated objects (replica groups) which are manipulated through a single, common interface.

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 (a well-known *manager process* on every node is responsible for creating servers). 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 */
}                                   /* objct1, objct2 passivated */
```

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. 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.

6.2: Replication Algorithms.

In Chapter 3 we saw how the State Machine approach to active replication specifies two conditions which must be met in order that all interactions with active replica groups maintain consistency of state amongst the replicas. The conditions were:

- C1: every non-faulty state machine replica receives every request (*agreement requirement*).
- C2: every non-faulty state machine replica processes the requests it receives in the same relative order (*order requirement*).

In the replication scheme which we have designed, condition C1 is met by the reliable group RPC mechanism (using rel/REL which was described in Chapter 4). The group RPC guarantees atomic delivery of requests i.e., requests sent from a (replicated) client to a replica group will be received by all functioning members of the group despite failures at

the client and at the replica group. The order in which the request will be received at each replica is not guaranteed, however. We have seen in Chapter 4 that a total order of messages is unnecessary for all applications, and therefore condition C2 is met at the application level by relying upon the serialisation property of atomic actions.

Such an approach can have two important performance related advantages: (i) it is usually possible to implement faster protocols for atomic delivery as compared to totally 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 designs to be described below indicate how object replication can be supported in atomic action based systems without the use of *order* preserving multicast communication primitives.

The Group RPC mechanism (GRPC) is used by invoking the *GRPC_call (G, ..., type ...)* on the replica group, where *G* is the group to which the request/reply is being sent. The termination conditions for a call are: *type = all*, meaning get replies from all functioning members of *G* and *type = one*, meaning a single reply would do. These termination conditions were described briefly in Chapter 4.

6.2.1: Overview.

One of the problems encountered in any replication protocol is ensuring that users of a replica group interact only with consistent members of that group (the Voting scheme described in Section 5.2.2 uses an update number associated with every replica which indicates whether the replica is up–to–date, and this needs to be inspected by every user of the replica group before it can decide whether or not to make use of a particular replica). Our scheme makes use of a database which maintains lists of only those (consistent) replicas which users can interact with (the database is called the *Group–View Database*). In our protocol, users acquire the list of consistent replicas (the *groupview*) from the database, and use this to interact with the replica group. This list consists of the location and name of each of the replicas in the group. After the clients have finished, if any of the

replicas are detected to have failed then this information is used to update the groupview held by the database.

Before communicating with a replica group, clients start an atomic action, within which all operations will occur. When the groupview has been obtained, the clients will issue requests on the replica group. All interactions with the replica group then occur by making use of the reliable group RPC mechanism, which ensures that all messages will be delivered to every functioning destination, despite failures. The serialization property of the atomic actions ensures that all interactions with the replica groups take place without interference from other clients.

During the interaction with the replica group, the clients assemble an *Exclude List*, which is a list of those replicas which have been detected to have failed (they did not respond to a given request). At commit time, this list is used to update the groupview held at the Group–View Database, so that the replicas whose names are in the exclude list are no longer in the group.

Note that we assume that it is possible to obtain the state of an object. This is necessary in order that recovering replicas can obtain the current state of the relevant objects before rejoining the group. We will use the *save_state* and *restore_state* operations which all persistent objects within Arjuna possess.

6.2.2: Failure Assumptions.

What will be discussed in the following sections is the design of a family of replication protocols for use in an object–oriented distributed system which supports atomic actions. Each protocol has been designed to function in an environment which conforms to a given set of failure assumptions.

- All of the replication protocols to be described assume that the nodes on which replicas will reside are *fail–silent* (Section 3.3.1).

However, the protocols differ in whether they can function correctly in the presence of network partitions: we say that a *network partition* has occurred if a functioning node is unable to communicate with some other functioning node. The term *virtual partition* will be used if this inability is temporary in nature, which is usually caused by network congestion and/or overloaded nodes.

6.2.2.1: Available Objects.

The first protocol to be presented (Available Objects) is based on the Available Copies database protocol (Section 5.2.1), and makes similar assumptions as described in Section 3.3.4.1.

- For this protocol to function correctly no network partitions can occur.

This assumption, combined with the assumption that all replicas reside on fail-silent nodes, means that the only reason a replica will not respond to a request is because that replica (node) has failed.

The current implementation of this replication protocol makes use of the reliable group RPC we have implemented (and described in Chapter 4). The group RPC uses the rel/REL reliable message passing protocol described in Section 4.5, which assumes that messages take at most a known bounded amount of time for reaching destinations. The rel/REL based RPC mechanism cannot tolerate virtual partitions, and therefore neither can the current implementation of Available Objects: nodes which are virtually partitioned will be assumed to have failed, and this can lead to inconsistencies between members of the same replica group.

However, if the reliable group RPC was implemented using a *Psync* style reliable message passing protocol (Section 4.3) then virtual partitions can be tolerated. *Psync* is a message history based protocol (described in Section 4.2) which does not make use of bounded delay assumptions and can therefore tolerate virtual partitions.

6.2.2.2: Voting Objects.

The second protocol described is a variation of the previous protocol, and is based on Weighted Voting (Section 5.2.2). This protocol can tolerate network partitions:

- if a rel/REL based group RPC is used, then virtual partitions can cause the affected nodes to be excluded from use.
- if a Psync based group RPC is used, then nodes which are virtually partitioned will not be excluded, and will still be available for use.

6.2.2.3: Primary Objects.

Unlike the first two protocols which use active replication, the final protocol makes use of passive replication, and is based on the Primary Copy replication protocol (Section 5.2.6). This protocol can also tolerate network partitions in the same manner as indicated above.

The replication protocols to be described can be split into two mechanisms: a means of ensuring a consistent view of a particular replica group between its users, and a means of preserving consistency of state between these replicas. The means of ensuring a consistent view of the members within a replica group is common to all of the protocols to follow, and is provided by the *Group–View Database*.

6.2.3: The Group–View Database.

To ensure that every user of a particular replica group sees the same group membership it is necessary for there to be available some method of obtaining the group view at any time. This is provided by the Group–View Database.

6.2.3.1: Replicated Object Information.

Information concerning every persistent replicated object which exists in the distributed system is recorded in the Group–View Database. The information maintained about the replicated object (group) includes a list of all of the objects which are members

of the group (including where they are located), called the *group list*, and (possibly) the type of service it exports (e.g., spreadsheet). This can be accessed by clients which require use of the service by either explicitly naming the replicated object itself (giving the *replica group name*), or perhaps by giving a service name (e.g., "Spreadsheet").

The group information also contains an *Exclude List*, which is a list of all those members mentioned in the group list which have failed i.e., whose states may be inconsistent due to failures. When a client requests the list of replicas which currently make up the replica group, the database uses the Exclude List as a mask, and only passes to the client the list containing the names of consistent replicas. Such a list is known as the *Available List*.

Finally, associated with every group is a *Use List* and a *Use Counter*, which indicates whether the group is currently in use. Everytime a client gains successful access to a replica group, the node on which the client resides and the client's name are added to the Use List as a <hostname, process_id> pair so that multiple users on the same node are identified uniquely, and the Use Counter is incremented. Once the client has terminated all interactions with the group, this counter will be decremented and the node reference removed.

All of the information held on behalf of a given group is collectively called the *groupview*.

6.2.3.2: Node Information.

Also maintained by the database is information on each node in the system. The information maintained is a *Node Exclude List*, which is a list of those replicas associated with that node, which have been detected to have failed and therefore will not be in the available lists, and an *InUse List* which is a list of all those replicas which belong to groups which are currently in use. This information is used by recovering nodes to determine which replicas need updating, and shall be explained later.

6.2.3.3: Example of Database Information.

In the example in Table 1–4, the replicated service *SpreadSheet* currently has 3 replicas located on nodes called *ulgham*, *semillon*, and *cabernet*. Of these three replicas, only the ones on *ulgham* and *cabernet* are in use, as the node *semillon* has failed. Currently 3 clients are making use of this group (indicated by the Use Counter field).

Object Name	Use List	Use Counter	Replica Name	Node Name	Exclude List
SpreadSheet	<ulgham, 0>	3	SpreadSheet1	ulgham	Available
	<ulgham, 1>		SpreadSheet2	semillon	Unavailable
	<murton, 0>		SpreadSheet3	cabernet	Available

Table 1–4: Group–View Database Entry.

The example node entry in Table 1–5 shows what would be visible for each node given the data in Table 1–4. Node *semillon* has an entry in its Exclude List for replica *SpreadSheet2*, whilst the other two nodes simply have an entry in their InUse List for their respective replica.

Node Name	Node Exclude List	InUse List
ulgham	-	SpreadSheet1
semillon	SpreadSheet2	-
cabernet	-	SpreadSheet3

Table 1–5: Node Entry for SpreadSheet Service.

6.2.3.4: Operations on the Group–View Database.

To prevent the Group–View Database from becoming an access bottleneck, we attempt to ensure that all accesses to it are brief. However, it is necessary to ensure that all accesses are concurrency controlled. If the client were to lock the entire Group–View Database, say while it is modifying a groupview, then this would prevent other clients from using the rest of the database. As such, we have structured the database so that it is made up of a collection of *group objects*. Each group object maintains the information on a single

replica group as described above. Whenever a client contacts the Group–View Database to access a particular replica group it acquires a lock on only the group object that it requires, instead of locking the entire database. This finer granularity of locking allows concurrent modification of separate groupviews.

All of the following database operations are invoked as top–level actions.

- *getview (objectname)*: this returns the current Available List. When this operation is called, it write locks the groupview from any other client, reads the Available List, and increments the appropriate *Use* counter after updating the *Use List*. The groupview is then unlocked and the call ends. All objects which are flagged as in use are placed onto a particular node's *InUse List* and removed when the client commits or aborts.

- *exclude* (<$objectname_1$, $excludelist_1$>, <$objectname_2$, $excludelist_2$>, ...): this operation allows a client to update the database's Exclude List with its own. The parameters passed are the name of the group which is to be updated, and the client's exclude list to use for the update. The operation first write locks the groupview and then updates the state of the replica group associated with *objectname*. When this has occurred the operation decrements the *Use Counter* and *Use List* for this particular client, and then unlocks the groupview. This sequence of events is performed sequentially for all entries passed as parameters. The update requires each entry on the client's Exclude List to be reflected at the database, both at the associated groupview's Exclude List and the individual node Exclude List, which must also be locked prior to its being updated.

- *include (objectname, hostname)*: this operation allows a recovering node to include a previously excluded replica (it can also be used to add completely new replicas). This operation is invoked after the recovering node has obtained the current state of the replicated object for the recovering replica (as detailed in Section 6.3.3). This operation can only occur when the group in question is not currently in use, as adding a new replica while the group is being used can pose problems with consistency between

replicas. This operation first write locks the groupview and then inspects the *Use Count* for the group. If some other client is currently using the group (the *Use Count* is not zero) then the operation must abort, unlocking the groupview, and the invoker of the operation must try again later. If the group is not in use then the Exclude List is modified to show that the replica is once again available (or the groupview is modified if this is a totally new replica). If this is a new object then it will be added to the corresponding replica group (or a new instance will be created if one does not exist). If the hostname is new then it is added to the database and the object will be added to any existing replica group. The operation then decrements the *Use Count* and ends.

- *remove (objectname, hostname)*: this operation is invoked by functioning nodes to remove objects associated with the *hostname* from the groupview. If the *Use Count* for *objectname* is zero then remove the entry (if any) for the *hostname* from the groupview, provided the Available List does not become empty; else return *trylater*. It is important for recovery reasons that the Available List does not become empty: since we shall be using replicas mentioned in the groupview to update recovering replicas, it is important that there be at least one replica always associated with a group otherwise recovering replicas will be unable to obtain an up–to–date state.

- *release (objectname, hostname)*: decrement the *Use* counter associated with *objectname* and remove the reference to *hostname* in the *Use List*. This operation is invoked by users of *objectname* which do not have exclude lists for the group but which do need to indicate to the Group–View Database that they have finished using the group i.e., it is used when an action commits or aborts.

- *recover (hostname)*: remove the entries for the *hostname* from all the *Use Lists* kept in the database. This is used by recovering nodes to remove references to themselves from the database. It is similar to *release* except that it is performed on all groups currently mentioned in the database.

- *status (replicaname, objectname)*: returns the current status of the given *replicaname* which is associated with the group *objectname*. This will be either *trylater* if the object group is currently in use, *not–modified* if the *replicaname* is not on an Exclude List, or *modified* if it appears on an exclude list. This operation is used during recovery to determine whether a replica needs to be updated before it can accept any further invocations.

Because all of the operations are mutually exclusive, requiring a write lock, and because this Group–View Database is used by every application level atomic action which accesses replicated objects, it is necessary to prevent it from becoming an access bottleneck. The design presented attempts to prevent this from happening by ensuring that the database is not locked for long durations.

6.2.3.5: Using the Database.

Making use of the example replicated service outlined in Section 6.2.3.3 we shall show how a client interacts with the Group–View Database. The client (*A*, say, on the node called *murton*) which wishes to make use of the replicated *SpreadSheet* service, interrogates the database for the *Available List*, by invoking the *getview("SpreadSheet")* operation. It will obtain a list of replicas which are available, and in this case the list will be of the form *[<ulgham, SpreadSheet1>, <cabernet, SpreadSheet3>]*. The *Use* counter and *UseList* associated with *SpreadSheet* will be updated to show that *A* has acquired the current groupview: the *Use* counter will be incremented to 4, and the *UseList* will have the hostname *murton* added to it.

Once the client has obtained the available list it invokes the *initiate* operation on all of the replicas mentioned in the Available List, as described in Section 6.1. The *initiate* operation returns to the client an address which the client uses to communicate with the replica group. The client can then invoke operations on the initiated objects. If a replica failure is detected then the client will assemble an exclude list on behalf of the object.

When *A* comes to commit, it will either call the *exclude* operation if it has a non–empty exclude list, or it will call the *release* operation as *release("SpreadSheet", murton)*. These operations will decrement the *Use Count* and remove the client's entry from the *Use List*, thus indicating that the group is no longer in use by this particular client. In this way, the data held at the database is always kept up–to–date with respect to committed actions.

Clients only remove entries from the *Use List* of groups and decrement the associated *Use Count* when their actions either commit or abort. As has been mentioned, recovering replicas can only re–join their groups when no other clients are using the groups i.e., when the *Use Count* associated with the groupview is zero. However, if a node fails before clients on it can decrement the *Use Counts* they have previously incremented, recovering replicas will never be able to re–join their groups. Thus, when a node recovers it must first invoke the *recover* operation on the Group–View Database to release any groups currently "locked" in this way. The node can then perform update operations on those out–of–date replicas which reside on it before invoking the *include* operation to add them to the groupviews held by the database.

6.2.3.6: Correctness Properties.

The groupview data stored in the Group–View Database satisfies 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 is 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 Group–View Database also satisfies 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 group 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 group 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.

6.3: Available Objects.

The first replication technique is based on active replication (Available Copies): every available replica attempts to execute the same request and then return the result to the invoker. The failure assumptions for this first protocol were stated in Section 6.2.2.

Our protocol functions differently for read and write operations: because our protocol ensures that all functioning replicas have identical states a client need not wait for more than one reply if it has invoked a read operation; a write operation requires all responses as the Exclude List must be updated with those replicas which do not respond. This is necessary to ensure that all replicas of a given group which a client makes use of will always have identical states; replicas which have failed will not receive and process operations which the other replicas receive, and these operations may modify the state of the replicated object. If the failed replicas were not excluded from being used by subsequent clients it is possible that their states could be used later, when the nodes on which they reside recover.

However, as we have seen, the object–oriented approach isolates the programmer from the actual implementation of an object by providing an interface through which operations are invoked on the object's state, and it is usually not possible to determine

simply from the interface whether or not a particular operation will modify the object's state.

6.3.1: Distinguishing Operations.

The method we shall use to distinguish the type of the operations is as follows: although the client may not know what effect a particular operation will have on a server's state, the server does. The client simply makes the call, but the server, as well as sending back the reply in the RPC also sends back enough information to indicate whether the operation was a read or a write. If it was a read then the client does not await any further replies (these will be discarded).

Another approach for distinguishing operations relies on there being some means available from the underlying system to actually distinguish between the types of operations. If the interface definition language in which the distributed applications are written qualifies operations as either read or write then this can be used e.g., in the Arjuna system, programs are written in C++ and it is possible to define *constant* functions in the C++ language. Such functions are assumed to not modify the state of objects they refer to i.e., are only of read type. This information could be used at compile time to modify the protocol to take into account the fact that no state changes will take place.

6.3.2: The Replication Protocol.

We shall now extend the example in Section 6.1 to show how a (replicated) client makes use of a server replica group in Arjuna. Consider a client object A which wishes to invoke an operation on some other object B (a SpreadSheet, say) which is currently passive. Both A and B are replicated. Initially the invoking object would start an atomic action within which all operations would occur. We will assume that object A has been activated, and all its active replicas belong to a group G_A. This sequence of events for A breaks down into the following (assume that GD is the Group–View Database object):

```
SpreadSheet B();                          /* Activate object */
[
      view = GD.getview("B");             /* Get Available List for object B */
      working = initiate(view);           /* initiate replica group */
      assemble_exclude_list(working);     /* Assemble Exclude List */
]

Atomic_Action Z;                  /* Action within which operations will occur */
Z.Begin();                        /* Start Atomic Action */

B.Set_Element();                          /* Invoke operation on SpreadSheet */
[
      replies = GRPC_call("B.Set_Element()"); /* Use group rpc for operations */
      assemble_exclude_list(replies);
]
Z.Commit();                               /* Commit Action */
[
      if exclude_list = empty
            GD.release("B", hostname);
      else
            GD.exclude(<"B", exclude_list>);
]
```

The course of events within [ ] are transparent to the user and are generated automatically by the stub generator and Arjuna libraries, but are shown here for clarity. We shall now describe the stages involved in more detail.

6.3.2.1: Replica Group Initiation.

The client object α ($\alpha \in G_A$) begins execution and starts an atomic action (Z). Before creating the replica 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). The client thus contacts the *Group–View Database* (*GD*) to obtain the addresses of the replicas it wishes to communicate with. The client then attempts to *activate* all copies mentioned in the database's Available List, making note of which replicas do not respond. If at least one replica was activated then the client action can continue to operate. More formally:

The client object invokes *initiate(...)* to activate the copies. Within *initiate* client objects manufacture a group name G_B to be used to communicate with the server replica group,

(all objects from the same client group will manufacture the same group name). 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. Within initiate every functioning α executes the following algorithm: it *individually* invokes the *getview(..B..)* operation as a *top level* action (which can commit independent of the invoking 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 $N = \{N_1 ... N_m\}$ 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. The manager process at each node has the capability of filtering out multiple create requests.

If N_j is functioning, a server 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 group RPC described in Chapter 4. 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. Replies from servers are received by every functioning α, which ensures that all members of G_A get an 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 the reliable group RPC mechanism.

After *initiate* returns, the client starts to assemble its Exclude List with the names of those replicas which did not respond.

6.3.2.2: Constructing and Processing Exclude Lists.

Because the Group–View Database does not actively keep track of which nodes in the system are available, the client action now has a (potentially) more up–to–date groupview. The Exclude List formed by the client is initially held locally to the client, and is updated

whenever a replica failure is detected. At the end of the atomic action the client will attempt to bring the database up-to-date with the new groupview (and any subsequent clients will see only the consistent replicas in this new groupview).

The Exclude List performs two functions in our protocol. The first is a mechanism for keeping track of which objects are available and consistent, and the second is a means of informing recovered nodes which objects need updating. During this discussion th as been an implicit assumption that there is one *logical* Exclude List per atomic action. If we consider that calls on replica groups can result in further calls on replica groups, it is possible to see that each client/server that issues a request on a replica group will form its own Exclude List based upon which other replica groups it may have communicated with.

To maintain the logical Exclude List (the one which will eventually be used to update the database) it would be possible to have servers propagate back to clients the exclude lists which they have assembled (this could form part of the reply which a server sends to the client). The clients could then use these exclude lists to modify their own and arrive at the final logical exclude list.

However, as was mentioned in Section 6.2.3.4 in the description of the database operation *include*, recovering replicas can only re-join a replica group once that group is not being used by other atomic actions. This can be determined by examining the *Use Count* associated with every replica group. If a node recovers, any replicas on it cannot be used by clients until they have been updated and the *Use Count* is zero (a full description can be found in Section 6.3.3). As such, even if a replica fails and recovers during the course of an action, it will appear as though it had never recovered. This means that we do not have to assemble the Exclude List during the execution of the action but after phase1 of the commit protocol has completed. As shall be shown, this property also means that a client need not wait for all replies from a replica group to arrive before continuing to execute.

6.3.2.3: Concurrency Control.

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) [Parrington 88a][Parrington 88b]; thus the user of an object is not explicitly responsible for obtaining a lock, rather the user 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 an example with two objects A and B, and associated object groups G_A and G_B as mentioned above Assume that the operation of B invoked by A requires a write lock; assume further that there is another object C (group G_C) that contains a call to B (which also requires a write lock). Thus G_C and G_A are both attempting to use G_B concurrently. 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 (G_C) has succeeded in its first invocation to G_B (meaning that G_B has been locked), all subsequent invocations from G_A (G_C) will give rise to identical responses from replicas – since no lock conflicts will occur (indeed, as we shall

see later, G_A (G_C) need not wait for all the replies for an invocation once it has write locked the group, 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. Note that it is important to realise that our concurrency control scheme will work correctly even if the clients of a server replica group are themselves replicated. Members of the same replica group can obtain write locks concurrently on the same server replica group because this cannot cause a conflict, as would be the case if the client replicas were from different groups.

Substantial performance improvements can be obtained by distinguishing between the two types of termination conditions for a group invocation:

- *type = all*, which requires replies from all the functioning members of the invoked group.

- *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. Using one of the techniques outlined in Section 6.3.1 we shall assume that the client making a call 'knows' whether the called operation is of type *read/write*, thus requiring a *read/write* lock. 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);

- (iii) the client is committing.

As was mentioned in Section 4.6.2, our reliable group RPC mechanism allows the number of replies required to be specified on a per call basis.

6.3.2.4: Issuing Requests.

When the client invokes the operation, the call is processed by every operational member of the server group and a reply sent from each. Because the client action knows who the members of the group are meant to be, it knows how many replies to expect (the group membership can shrink at any time due to failures, but cannot increase while the group is in use). If at least one member of the replica group remains operational then the action can continue.

The client can then issue requests on the replica group by making use of the *GRPC_call()* described in Section 6.2. The return code from this call will be one of the following:

- *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). This indicates whether the replica group has already been locked by a separate client in a conflicting mode.

6.3.2.5: Committing Actions.

When the client action comes to commit it must do two things: it must commit the object group(s) it has used, and update the view of the group(s) in the Group–View Database. If during the commit further servers are detected to have failed then the Exclude List is again modified. The Exclude List is essentially a list of those replicas which are now in an inconsistent state with respect to other replicas from the same group. However, if read operations were only performed on a particular replica group, then failed copies are still in a consistent state. In Arjuna, when the client action comes to commit it may inspect the locks which it acquired on the replica group(s) and remove from the Exclude List any reference to replicas which were simply read, finally obtaining a list of only those failed replicas which are now out–of–date.

During the first phase of the commit protocol, the client action informs the servers that they should *prepare* to commit. Before the commit decision can be made the Group–View Database must be updated. This is run as a separate top–level atomic action, where the client action transmits its Exclude List to the database, and the database updates the Exclude List associated with that group accordingly. If this Group–View update fails for whatever reason, then the client action must abort, once again bringing all replicas (both

active and failed) back into consistent states. This sequence of actions can best be viewed by examining Figure 6–1.

If the Group–View update succeeds then the client action can commit.

In a conventional 2–phase atomic action system if an object which was *touched* during the lifetime of the action does not respond during the phase1 prepare phase then the atomic action aborts. However, by introducing the concept of replica groups the definition of when an action can commit changes. In our replication strategy, an action may commit as long as there is *at least one* replica still active in the group by the end of phase1 (the failed copies having been added to the Exclude List).

Assuming that no groups have failed, then once the replies have been gathered, the coordinating action attempts to update the Group–View Database by calling the *exclude* operation and transmitting its Exclude List to it. If this action fails then the user's action must also fail. The client action then moves on to phase 2 of the protocol. If phase 1 was successful the client will signal the objects to save their state to stable storage, otherwise the client will abort the action and signal the objects to do so as well.

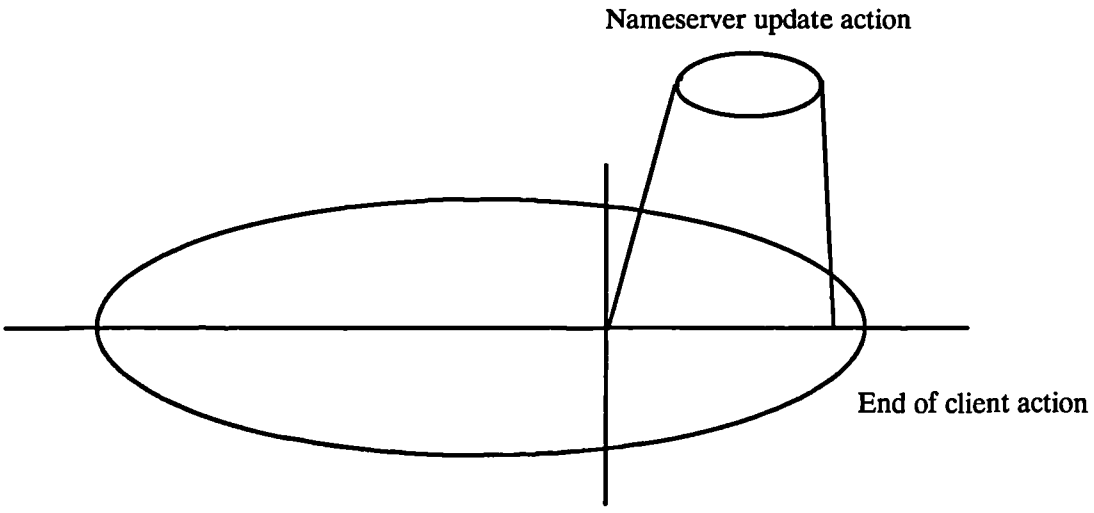


Figure 6–1: Committing the Exclude List.

The relationship between the two atomic actions (application and update) is that the application action can only commit if the database update action also commits, but the

database update action can still commit in the absence of an application action commit (because failed replicas have been excluded from the groupview the protocol guarantees that all of those replicas which appear in the available list will be in a consistent state).

6.3.2.6: Commit Optimizations.

If the assumption is made that the Group–View Database is always available, then there will never be a situation in which the client atomic action must abort because the database update fails. Therefore, it would be possible to use a separate thread of control to complete the second phase of the commit protocol at the client, so that the client code can continue to execute as soon as the first phase of the commit protocol has completed successfully. This main thread would contact the database and update it with the client's Exclude List after phase 1, creating another thread to finish phase 2.

The saving of object states on the disk can consume some considerable percentage of the time taken for an action to commit. If we assume that each node is equipped with an uninterrupted power supply (UPS) powerful enough to allow it to write all of its volatile memory to stable storage in the case of a power failure (i.e., stable ram), then it would be possible for the replicas to delay saving their states. During the commit phase the object does not immediately save its state to stable storage, but responds to the action coordinator as though it had saved its state. The replica can then save its state after the action commits. This would lead to a faster commit time for atomic actions. The stable ram ensures that if the node fails the state will be saved.

6.3.2.7: Termination of Replica Groups.

When the objects which comprise a replica group go out of scope for a particular client then the *terminate* (..G_B..) primitive is called to passivate the objects by destroying the server processes. This uses the *GRPC_call*(..G_B..) for terminating G_B.

6.3.3: Node Recovery.

We must ensure that any recovering object does not service a request until it has been brought up–to–date with other replicas in the group. Let us consider what happens at a recovering node which contains a (formerly unavailable) replica. The node must ensure that every object which belongs to a replica group has its state updated if necessary. As part of the crash recovery mechanism, a node would first ensure that it is able to run distributed actions on its own behalf but continue to refuse to run actions for other remote objects (so that the node appears to be still unavailable until it has fully recovered).

To start with, a recovering node calls the *recover* operation of the Group–View 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 (i.e., the Use List is empty). Thus, node *i* executes the *status* operation for every replica it is maintaining; if the response is *not–modified* then no further update 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*. This involves gaining exclusive access to the groupview during the update operation, and so the recovering node attempts to write–lock the members of the group for the duration of the update. Once the group is locked, the recovering node can request the current state from the group members, using the first state which it receives since they will all be identical.

Note that as an efficiency measure, the recovering node can request from the database its *InUse List* as well as the Exclude List, so that it can determine locally which replicas

need updating, and which replicas may need updating when they are no longer being used by other clients.

When the recovering replica has been brought up–to–date in this way the *include* operation is invoked and the write lock obtained by *get_exclusive_groupview* is released. 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.

Between the time a failure of a replica is detected by a client (C) and this failure being reflected at the Group–View Database, there is a finite space of time in which the failed node could recover. When it does recover it will not find any reference to itself in the Exclude List (since the updates have not been written), and so will assume every object is up–to–date. However, since the *Use Count* associated with the groupview will not be zero because C is using the group, the recovering node cannot let any of these replicas be accessed from other remote nodes until the database groupview has been unlocked and the recovering replicas have been brought up–to–date.

This could lead to the situation where a recovering node which possesses many objects which are up–to–date refuses to accept requests because a minority of its objects are currently in an indeterminate state. This can be resolved in two ways:

(i) In Arjuna, there is a manager daemon which is responsible for activating servers to handle calls to objects. This can be modified so that it can filter out requests for those objects whose states have not been up–dated. When these objects finally recover, the manager can again accept requests to activate them.

(ii) When a server is created by the manager daemon it needs to initialise its state. By removing access to the present (out–of–date) state, the server will be unable to initialise itself and this can be translated into a failure of the server. Such a failure can be

communicated to the client which requested the server initialisation as though the server node has failed.

Either of these methods will allow those up–to–date objects on a recovered node to be accessed, while at the same time maintaining the illusion that the out–of–date objects have remained in a failed state.

The Group–View Database and the Exclude List are efficient methods of ensuring that replica consistency is maintained in the presence of failures. The Exclude List is also an efficient method of ensuring that recovering nodes do not have to execute update operations on all objects held at the node (which could number in the thousands) but can instead execute update operations on only those objects which need updating. This means that a node can recover quickly, and objects can re–join their respective replica groups.

6.3.4: Making The Group–View Database Highly Available.

The replication schemes to be discussed here require 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 Group–View Database is implemented as an Arjuna class *group–view* and n ($n > 1$) instances have been created to get an n–replicated object, *group–view*. Being an Arjuna object means that *group–view* 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 *group–view* is kept locked only for short durations (this is important, otherwise *group–view* could become an access bottleneck).

We assume that every node 'knows' the locations of the n copies of *group–view*. 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 *group–view* 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 *group–view* provides one more operation *getstate(...)* which returns the current state. A recovering node N_i containing a copy of the Group–View Database 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. Until the recovering database replica has been brought up–to–date it cannot be used by clients.

6.3.5: Implementation.

A trial version of object replication has been implemented in Arjuna and the results obtained are promising. The Group–View Database (currently unreplicated) has been implemented with the operations *getview*, *include*, *exclude*, and *remove*. The action mechanism has been modified to construct Exclude Lists should failures of the replicas be detected. The present implementation does not provide automatic transparent replication as these mechanisms have not been integrated with the C++ stub generator. This will be achieved in the future implementation.

The problems with distinguishing the type of an operation in an object–oriented environment have been described in Section 6.3.1, and our implementation currently uses the method which has the server perform the requested operation and then return, along with the reply, an indication of whether the request modified the state. However, in the current implementation all operations are assumed to be of *type=all*, and so the optimizations which are available for *type=one* have not been tested.

The locking policy for replica groups as described in Section 6.3.2.3 has been implemented and tested. When the remote objects perform the operation requested, they

return information to the client indicating whether or not the group has been locked on its behalf. In the current implementation, if the entire group has not been locked then the action should abort and try the operation later.

To test the replication implementation, an unreplicated client was written which would interact with a remote replica group and perform operations on the group within an atomic action. Each replica within the replica group was of type *ActInt*, as shown below:

```
class ActInt
{
        int Element;
public:
        ActInt(int& v);
        ~ActInt();
        int Set(int v);
        int Get(int& v);
};
```

ActInt is a simple object which has only one integer state variable, *Element*, and provides operations to read and write that variable (*get* and *set* respectively). Each of the operations provided executes within an atomic action.

The commit phase of the atomic action can be represented as follows:

```
AtomicAction_Commit()
{
      if (prepare() == PREPARE_NOT_OK)
            phase2_abort();
      else
            phase2_commit();
}
```

The *prepare* phase of the protocol attempts to contact every object used in the lifetime of the commiting action, and in a non-replicated situation if some objects do not respond then the prepare phase returns PREPARE_NOT_OK and the action proceeds to abort. Only if all objects respond can the action proceed to *phase2_commit*. When replication is used, the prepare phase of the action is modified so that it fails only if a replica group fails

i.e., one response from a replica group will still allow the action to commit despite other failures in the group.

The timings taken for this were made at the commit level of the atomic action. In the non–replicated case, a client interacted with a remote instance of the *ActInt* object and then proceeded to commit. When a failure was required it was arranged so that the remote object would fail before the commit phase, but after the last time the object was used by the client to perform any operations.

Status of remote ActInt Object	Time taken to commit/abort
Operational	0.57 seconds
Failed	30.40 seconds

Table 1–6: Non–replicated commit timings.

The first timing of 0.57 seconds takes into account the time taken by the remote object to save its state to stable storage and to then reply to the action coordinator. The difference of approximately 30 seconds between the two timings is because when the commit phase of the action does not receive a reply from the *ActInt* object it attempts to contact the object again to ensure that the original message was not lost due to buffer overflow or an overloaded node. This takes 15 seconds (3 calls in total with 5 second timeout values each) and when no reply is finally received the action determines that the remote object is no longer operational. The action then proceeds to the abort phase and it attempts to contact every object used in the action to inform them that they should abort. The action commit phase tries very hard to ensure that all objects will be informed of the abort decision, and so when it cannot contact the failed object during abort it again tries 2 more times, with timeouts of 5 seconds each i.e., the abort phase takes another 15 seconds.

When an *ActInt* replica group is used in place of a single object, the timings produced are different (the replica group had two members, and only one of them was made to fail):

Status of remote ActInt Object Group	Time taken to commit
Operational	1.07 seconds
One replica failed	15.70 seconds

Table 1-7: Replicated commit timings.

The additional time taken to commit the group when all members are operational is a result of having to communicate with twice the number of objects than previously, each of which must save its state to stable storage before it can reply. The second timing shows that although the first phase of the commit protocol takes approximately 15 seconds because of the failed replica, the action then proceeds to the *phase2_commit* stage. The fact that one replica has failed is noted, and it is excluded from this second phase commit, so no further overhead is imposed. Writing the Exclude List to the Group–View Database takes a few tens of milliseconds, and has not been separately shown.

The commit time in the presence of failures can be made to approach the failure free time by introducing concurrency in the commit protocol as was indicated in Section 6.3.2.6. This will be done in future implementations.

6.4: Tolerating Network Partitions.

The Available Objects replication protocol previously presented cannot work in the presence of network partitions. This is because it is not possible for a client to differentiate between a failure of a replica and a partition which prevents the client and replica from communicating. This could lead to clients having different groupviews for the same object group and this in turn could lead to inconsistencies arising in the states of the replicas within those groups. However, it is possible to modify the Available Objects protocol so that it can function correctly in the presence of network partitions.

6.4.1: Protocol Overview.

We shall initially assume that there is only one copy of the Group–View Database, although we shall later show how it too can be replicated. At the start of an atomic action

clients use the database as in the Available Objects protocol to obtain the groupview. The clients then *initiate* the replicas mentioned in the groupview. The names of any uninitiated replicas are recorded in the exclude list. These replicas are not invoked in all subsequent calls.

For all subsequent invocations, the clients continue to assemble/modify their exclude lists if further failures are detected. They also use the exclude lists as a filter to ignore any replies from out-of-date replicas or replicas which reply to requests subsequent to being detected to have "failed", as these replica potentially have out-of-date states also. At commit time the clients start the commit protocol (phase 1), modifying their exclude lists if necessary, and as long as one replica remains operational then the client action can continue to commit.

After phase 1 of the commit protocol has ended a client has an exclude list which contains the names of all of those replicas which did not respond to *all* of its requests. Because network partitions could have occurred, and because some replicas may have been slow in responding to some clients rather than others (virtual partitioning), it is possible that replicated clients have different exclude lists.

As in Available Objects, after phase 1 of the commit and before phase 2, the clients contact the Group-View Database, invoking the *exclude* operation. If any clients cannot contact the database then they must abort. If the network is partitioned then only those clients which can contact the Group-View Database can be allowed to commit. In this way, only those actions which occur within the same partition as the database will be allowed to change the state of a replica group. When the database has received all of the exclude lists it simply takes the intersection of all of them to get the *group exclude list*: this is a list of all of those replicas which did not respond to *any* of the committing clients during the action.

6.4.1.1: Action Divergence.

Chapter 4 showed how the states of replicas within the same group could diverge if message delivery guarantees could not be met i.e., if some members of a replica group received messages which other members did not receive. That was the reason for developing the reliable group RPC mechanism. As we shall see, for operation invocations this protocol functions similarly to the Available Objects protocol, and as long as one replica of a group is available an action can commit. However, this protocol functions differently when we consider what happens at replicated clients when they come to make a commit or abort decision.

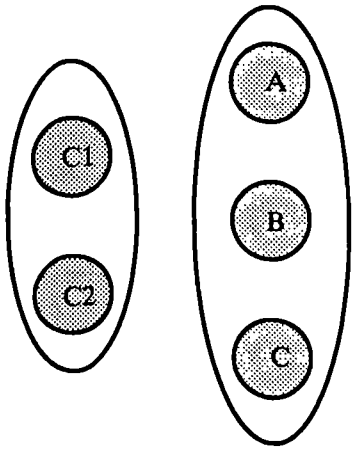


Figure 6–2: Divergence of replicated actions.

Figure 6–2 shows replicated clients *C1* and *C2* and the server group they have been communicating with whose member replicas are *A*, *B*, and *C*. If both *C1* and *C2* communicate with all members of the server group and therefore receive the same sequence of messages then this protocol will function in exactly the same way as the Available Objects protocol. If *C1* commits then *C2* will also commit and vice versa.

Suppose that *C1* decides that replica *C* has failed, and *C2* decides that replica *A* has failed. Thus, *C* will appear in *C1*s exclude list, while *A* will appear in *C2*s exclude list (assume that replica *B* does not fail). Because both client replicas have a common server

replica (*B*) then both clients will again receive the same sequence of messages and will both either commit or abort.

Now assume that *C1* can communicate with all members of the server replica group, but because of a network partition *C2* determines that *A*, *B*, and *C* have failed. *C2* will abort its action, and could then invoke operations on another group. Because *C1* has not come to the same decision as *C2* the states of the two clients could diverge.

Thus, different replicas of the same replica group could come to different decisions about whether or not to commit an action. Therefore it is necessary to maintain the consistency of those replicas mentioned in the database despite such possible inconsistencies of commiting actions. In order to do this, we allow commits to proceed as in the Available Objects replication protocol, but aborts occur differently: if an client has to abort then after the abort has occurred it takes no further part in computations. Therefore, in the situation shown in Figure 6–2 after the abort *C2* will invoke no further computations.

How this replication protocol affects the committing and aborting of actions will be described in more detail in Section 6.4.2.7.

6.4.1.2: Example.

Consider the situation shown in figure 6–3, where we have multiple client replicas (*A*, *B*, and *C*) communicating with server replicas (*D*, *E*, and *F*). Because partitions can occur it is possible during the lifetime of an action that each client replica can obtain a different view of the server group membership. For this example we shall assume that client *C* believes servers *E* and *F* have failed when the other clients believe them to be alive. As soon as a client determines that a server replica has failed it refuses to accept any further responses from it for the duration of this action. In the above case, clients *A* and *B* see the same groupview containing all of the replicas. Client *C* has decided that servers *E* and *F* have failed and so will have their names in its Exclude List.

When the clients commit they will transfer their Exclude Lists to the Group–View Database, which will construct the intersection of all of them and come up with the final Exclude List to use. If client *C* fails before contacting the database then only *A* and *B* will commit and so servers *E* and *F* will not be excluded anyway. Since all of the replicas responded to all of the requests issued by *A* and *B* they must possess identical states. If all of the client replicas respond (i.e., *A*, *B*, and *C* commit) then the database will take the intersection of their exclude lists, which in this case will mean that none of the server replicas will be excluded since they all responded to at least one of the client replicas.

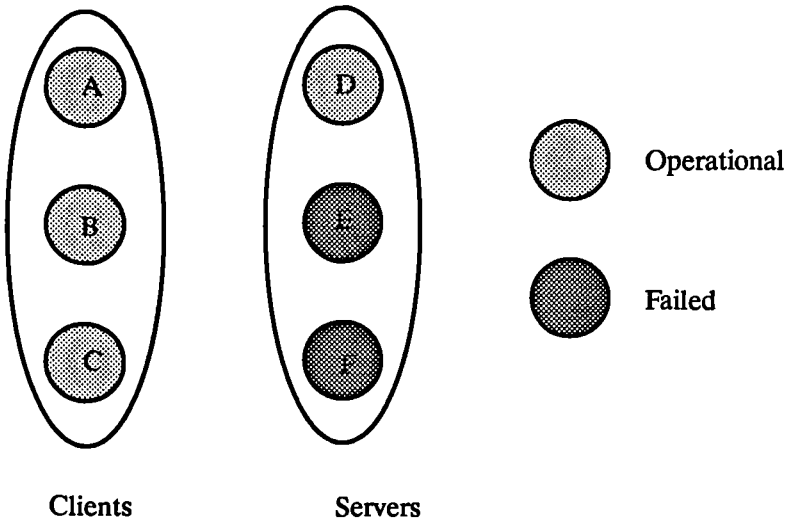


Figure 6–3: Diverging Exclude Lists.

As we shall see, although this algorithm can tolerate network partitions, in the case where partitions do not occur it executes less efficiently than the Available Objects protocol.

6.4.2: The Algorithm.

The Group–View Database is used in this protocol to ensure that all clients obtain the same view of a given replica group at the time of initiation despite communication failures such as network partitions. The description above assumed that there was only one copy of the database, but it is possible to replicate the database to improve availability. The

replication of the database can be handled in a similar way to that described in Section 6.3.4 except that users of the database must be able to contact a *majority* of the database replicas in order that all users obtain the same groupview information despite partitions.

By modifying the Available Objects replication protocol so that a majority of Group–View Database replicas must always be contacted, it is possible for the original scheme to then tolerate network partitions. The reliable group RPC which is used by Available Objects guarantees delivery of messages to all functioning replicas in the absence of network partitions. The need to contact a majority of the database replicas guarantees that only those client and server replicas which reside in the majority partition will be able to operate if a network partition does occur. Any other replicas will be excluded from the groupview at commit time and will be forced to abort.

However, it is possible to remove the need for a reliable group RPC mechanism entirely and still guarantee replica consistency in the presence of failures (this allows a cheaper communications protocol to be used). In the description to follow we shall assume the group RPC mechanism does not guarantee reliable delivery of messages and we show how replica consistency is maintained in the presence of failures.

6.4.2.1: Object State.

For this protocol to function correctly it is necessary for users of a replica group to be able to determine whether a replica has an up–to–date state. In the Available Objects protocol all objects mentioned in the Available List were guaranteed to have the same state, and all functioning replicas would continue to have identical states. As it is no longer now the case that every functioning replica in the groupview will be up–to–date, each object must possess an *update–count* as part of its persistent state which indicates how recently is was modified. This count is incremented after each successful top–level action commit, and is communicated to all users of the object in the object's replies.

6.4.2.2: The Group–View Database.

The Group–View Database is almost the same as that presented in the previous algorithm, but with some modifications. Every copy of the database maintains a version number (the *update–count* described in Section 6.4.2.1) associated with every *groupview*, which indicates how recently the groupview was modified. Clients use the most recently modified groupview which is agreed upon by a *majority* of the database replicas i.e., the most up–to–date groupview. If a majority of database replicas cannot be contacted then the client cannot continue with this action.

The *Use List* for each object group is also modified so that it now includes an *action id* associated with every client mentioned in the *Use List*. Replicated actions possess the same action ids, which are unique across different replicated actions. Exclude lists are associated with the actions within which they were assembled and the action id is included in all operations invoked on the database. The action id is used so that the intersection of exclude lists only occurs for exclude lists which were assembled in the same replicated action.

All operations on the database must be carried out on a majority of the database replicas. Any client who fails to obtain a quorum of database replicas must abort. Therefore, no client replica can communicate with a server replica that other client replicas do not know about from their initial groupview.

As a computation progresses, it is possible for members of the same replicated client group to have a different view of the server replica group, and therefore possess different Exclude Lists. The Group–View Database takes the intersection of all Exclude Lists given to it by replicated clients. In this way a consistent view can be obtained because all server replicas not in this composite Exclude List have been seen to have failed by all of the replicated clients. The database only merges those exclude lists which were from the same replicated action i.e., that possess the same action id, as merging exclude lists from separate clients cannot guarantee that the replicas mentioned in the subsequent Available

List will have identical states. When the merging of exclude lists is taking place it is not possible for new users to acquire the groupview. This is to prevent new users from obtaining an out–of–date groupview (one which contains references to replicas which are available but possess out–of–date states).

However, the merging of exclude lists can only occur swiftly if all replicated clients give the database their exclude lists at approximately the same time. If the database has to wait for all members of a client group to present it with exclude lists then the groupview being updated could be locked for long durations. One way to avoid unnecessary locking of the groupview would be that once the merging operation is begun the database will only wait for client exclude lists until another request for this groupview is received (either a read or update request), at which time it will impose those exclude lists it has received onto the groupview. This could result in replicas being excluded which possess consistent states, but still ensures that those replicas mentioned in the Available List are consistent.

Upon contacting the database, if a client (A) finds that a replica (B) it has been communicating with has been excluded by another client which is not a member of A's group, then A must abort because B could (potentially) have an out–of–date state which A has been using. A can then acquire the new groupview and try again. This prevents *different* client groups from updating the state of disjoint subsets of the same server replica group.

6.4.2.3: Example.

Consider the interaction of clients and replica group shown in Figure 6–4. Clients *C1* and *C2* are separate clients (not members of the same replica group) and contact the database and obtain the same groupview for the server group (A, B, C, and D). However, the network has become partitioned in the manner shown, which prevents *C2* from being able to communicate with A and B, and prevents *C1* from being able to communicate with C and D. When *C1* and *C2* initiate the server group they thus construct their exclude lists, and because they can communicate with at least one replica in the group the clients can continue to execute.

If we assume that both clients can still communicate with the Group–View Database, then at commit time they pass their exclude lists to the database. Because the clients are not members of the same replica group it does not make sense to merge their exclude lists (if *C1* had incremented the states of the replicas it communicated with by 2 and *C2* had decremented the states of its server replicas by 1, say, then the partitioned replicas have inconsistent states; however, the intersection of their exclude lists would not exclude any replicas). Therefore, no intersection of exclude lists is taken, and to ensure that the replicas mentioned in the Available List are consistent, the first client to contact the database (*C1*, say) will be able to impose its view of the group onto subsequent clients. When *C2* contacts the database it will determine that the server replicas with which it has been communicating have been excluded, and so it must abort.

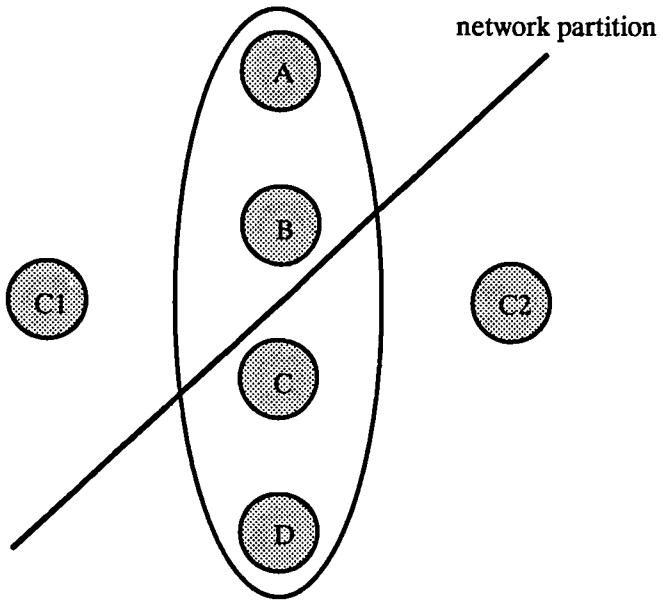


Figure 6–4: Groupview Divergence.

6.4.2.4: Replica Group Initiation.

At initiation time, the clients attempt to obtain the groupview from the database. Those clients which cannot obtain replies from a majority of the databases cannot continue with this initiation and must abort. However, those clients which have obtained

the correct groupview information can continue and activate the objects mentioned in the Available List. As part of the *getview* operation, the Group–View Database also returns the *update–count* for the group in question so that clients can detect out–of–date replicas.

Unlike the Available Objects protocol where it was possible to assemble the Exclude Lists at the end of the action as was shown in Section 6.3.2.2, in this protocol it is not possible to do this. This is a result of the difference in the failure assumptions made by the two protocols. Because we assume that it is now possible for network partitions to occur and for replicas to fail to respond to some clients whilst responding to others, replicated clients can possess different views of the same group (a replica which does not respond to one request could respond to subsequent requests). This could result in replicas from the same group possessing different states as they receive and respond to different requests. To prevent this, each client must assemble its Exclude List immediately when the *initiate* primitive returns. Further, the Exclude List will have to be constructed at each operation invocation, as described in Section 6.3.2.2.

If at least one replica responds to the *initiate* request then the action can continue. It is neither necessary that all members of the same client group *initiate* the same number of replicas, nor that they *initiate* the same replicas or some overlapping subset.

6.4.2.5: Operation Invocations.

When invoking operation on replica groups the clients use their Exclude Lists as a *filter* when accepting replies from replicas. Replicas mentioned in the client's Exclude List could potentially have out–of–date states and so the client will continue to refuse to use them even if a replica which appeared to have failed during one call resumes communication for the next call. A group will be considered to have failed if *all* of its members are mentioned in an Exclude List for an action. Unlike the previous replication protocol, all invocations made on an object group must be of *type* = *all*.

If a means exists of informing clients that the operations a "failed" replica missed were only "read" operations, then it would be possible to allow the client to remove the replica

from its Exclude List and resume accepting results from it. This is an optimization to maintain a level of availability; during the course of an action the exclusion of such replicas is logically correct, maintaining consistency of the replicas.

When a response from a replica is received, the client first determines whether the replica has an up–to–date state (by comparing the *update–count* received from the Group–View Database with the *update–count* returned by all replicas as part of their reply). If the replica has a lower *update–count* then it potentially has an out–of–date state and the client will add it to its exclude list and treat it as though it has failed. If a client receives a reply from a replica with an *update–count* greater than the one it currently possesses then it should abort this action as this client has an out–of–date groupview.

6.4.2.6: Locking of Replica Groups.

A client group will attempt to lock all of the replicas in the server replica group which responded to *initiate*, and if a *majority* do not respond then it must consider the operation a failure and unlock those replicas that did respond, and possibly retry later. Any client which does lock a majority of the replica group, either initially or subsequent to another client group aborting a lock operation, can proceed to use the group. Those replicas which do not respond (because they have failed or are simply too slow in replying) will be added to the Exclude List and will be ignored if they subsequently do respond.

Note that it is not necessary for different client groups to lock a majority of all of the replicas mentioned in the groupview, only a majority of the replicas which were initiated. This can lead to different client groups using different members of the same server replica group concurrently (and possibly in a conflicting manner). However, as was shown in Section 6.4.2.2, this conflict will be detected at commit time when the Group–View Database is contacted, and only one client group will be permitted to commit; the others will have to abort.

6.4.2.7: Committing Actions.

When a client top-level action comes to commit, it enters phase 1 of the commit protocol and informs the members of the replica group to *prepare* to commit. Those replica group members mentioned in the Exclude List are not sent any messages. The responses to the prepare message are used to modify the Exclude List, as was shown in Section 6.3.2.5. The client now has an up-to-date Exclude List and can update the Group-View Database with this information, also incrementing the *update-count* for the replica group. To do this it needs to communicate with a *majority* of the database replicas. If a majority do not respond favourably then the client must abort the action (or retry later). When the database has been updated, the client can *commit* the replicas not mentioned in the exclude list.

6.4.2.8: Aborting Actions.

A client must abort any action within which it cannot contact a majority of the Group-View Database replicas, or if the client finds that an entire server replica group has failed. To maintain consistency between replicas executing the same replicated actions it is further necessary that a replica which aborts an action should also be terminated i.e., it can take no further actions once it has aborted and so should terminate. This is a direct result of the failure assumptions we have made about the distributed system: no one replica can be certain that the operations it is about to invoke are being carried out by other replicas in its group. Messages can be lost, and as a result some replicas may be in a position where they have to abort the action, whilst others may not be. If a replica which aborts an action were to be allowed to continue operation it might then take some course of action which is different to the other replicas in its group, resulting in state divergence. By insisting that the replica should die we are preserving consistency between replicas.

Atomic action aborts within this replication protocol are assumed to be purely local aborts i.e., no messages are transmitted to other objects informing them that they too

should abort. This is necessary because these objects may be shared between a replica group some members of which may not require the action to abort.

However, if all of the replicas have come to the same decision to abort the action then it would be correct to allow them to continue to execute. It is therefore possible to execute a protocol between the members of the replica group to determine whether or not the aborting replica should also die. Such a replica would attempt to communicate with all replicas in its group and if they can all be contacted and they all agree to abort, then it would be possible for the replica to continue to operate after the action has aborted.

6.4.2.9: Uncommitted Replicas.

During phase 1 of the commit protocol, those server replicas which are on a client's Exclude List may not receive any message from the coordinating client. If this client was itself replicated, and all of the client replicas also assumed that the server replica had failed then it may receive no indication of the state of the action.

It is necessary to ensure that those replicas which were excluded from committing but were actually operational will eventually abort, undoing any state changes which did occur before they were considered to have failed, and then updating their states before rejoining the replica group. There are two ways of ensuring this.

In an atomic action system, the host node of a client which acts as the coordinator for a top–level action maintains some information which enables a recovering node to determine the outcome of that action and either commit (if it failed during the commit phase) or abort. In the case of replication, the information which should be stored needs to be modified so that to those replicas mentioned in an Exclude List it appears as though the action aborted, but to other replicas it appears as though it did commit. This can be done by having a groupview associated with any record information about the outcome of a given action, and if a particular replica finds that it is not mentioned in the groupview then it aborts the action.

Another method of ensuring excluded replicas eventually abort uses the Group–View Database again. Since these replicas have been excluded at the database, this prevents them being used by any subsequent clients. When these uncommitted replicas enquire as to the outcome of an action, they can first check the Group–View Database to determine whether or not they have been excluded. If they are mentioned in an Exclude List then the replicas will perform the same sequence of steps that recovering replicas must go through before being able to rejoin the replica group. If the replica cannot contact a majority of the database replicas then it should abort i.e., it should die.

6.4.2.10: Recovering Nodes.

The procedure for node recovery in this protocol is the same as that described in Section 6.3.3, except that such nodes must again vote on the Group–View Database i.e., they must be able to communicate with a majority of the database replicas before being able to update their local replicas.

6.4.2.11: Updating of Replicas.

If a client replica detects a server replica which is out–of–date, due to the client not having received replies from the replica for some time (i.e., the replica is in the client's Exclude List) then it can trigger it to carry out update actions. If a replica does not receive any invocations from a client for a period of time it could also inspect the GroupView Database's Exclude List for its own entry. If the database indicates that this replica has been excluded then the replica can attempt to update its state and rejoin the group.

6.4.3: Assessment.

The advantages of this protocol have already been mentioned: it can tolerate network partitions (including virtual partitions). Unlike the Weighted Voting protocol described in Section 5.2.2 which can also tolerate network partitions, it is not necessary for a majority of server replicas to be always available: as long as one replica is available an action can continue. However, this functionality has been obtained by paying a cost in performance.

Because it cannot be guaranteed that replicas which do not respond have failed, it is possible that functioning replicas of the same group can possess states which diverge. As a result of this, it is no longer possible for clients to be able to continue executing as soon as they receive a single reply from a replica not mentioned in the client's exclude list. The client must wait for the time allocated to a particular call and collect all responses, modifying its exclude list accordingly. Section 6.2 described the two termination conditions which exist for the reliable group RPC mechanism i.e., *type* = *all*, and *type* = *one*. For this protocol to function correctly, the RPC must always operate with *type* = *all* as it is important for clients to know which replicas have received requests and which have not. In the Available Objects protocol it was guaranteed that if one functioning replica received a given request then all functioning replicas would receive the request. This can no longer be guaranteed.

We have already seen that clients of the same replica group can have different views of the same server replica group (something which cannot occur in Available Objects). Because of this it is possible that some clients will assume the server group has failed whilst others are still invoking operations on the members of the group. The clients which determine the group to have failed must abort as was described in Section 6.4.2.8. Thus, with this protocol client availability is lost.

As was shown in Section 6.4.2.3, if partitions occur it is possible that different client groups can communicate with different members of the same server replica group. It is further possible that they will then be able to communicate with a majority of the Group–View Database replicas to update the groupview for this server group. The database will only allow one of the clients to impose an exclude list on the group (the first one to invoke the *exclude* operation) and the others will be forced to abort, undoing any work they have done.

6.4.4: The Update Daemon.

Because of the assumptions made about the communication medium and the nodes in the system, it is now possible for a call to fail to reach a given object even though the node is functioning (it could be severely overloaded). In this case the replica would eventually be excluded.

However, we now have a functioning replica which will never receive any requests. This can be solved by having a Site Manager (SM) whose job is periodically to inspect the Exclude List held by the Group–View Database on behalf of the node on which the SM resides, and if it finds a reference to any replica(s) which are known to be functioning then it can force them to run an update and then rejoin the group.

6.4.5: Group–View Database Update.

Because Group–View Database replicas are never excluded, this could mean that out–of–date copies of the database will never be brought back up–to–date. However, if a client finds that it can communicate with a Group–View Database replica which is out–of–date, then it can trigger the replica to perform an update action at the next convenient (quiet) point, as mentioned in Section 6.4.2.11. Out–of–date database replicas are recognised because, whenever a groupview is modified, the *update–count* is incremented at a majority of the Group–View Database replicas, and so clients can obtain the most recent value. Any database replica with a value less than this must be out–of–date as far as this groupview is concerned. The database replica can then perform an update action, which can update either the entire database state, or can update only that groupview which has been found to be out–of–date (this is an optimization allowing the database replica to rejoin the group much quicker than would be the case if the entire state had to be updated).

6.5: Passive Replication.

In a passive replication scheme a service is replicated N times, but of those N copies only one (the *primary copy*) communicates with client processes. This primary copy executes all requests and replies to the client, while at the same time distributing the new state to the remaining replicas, whose sole duty is to monitor the state of the primary in case it should fail. If a failure is detected then the remaining copies must elect a new primary to take over.

Before any reply returns from this primary all other replicas will be brought up–to–date by the primary (or excluded from its view). The response to the client is atomic in the sense that if the primary responds then all of the passive functioning replicas have been updated, and if no response is made then none of the passive replicas have had their states modified.

One aspect of passive replication that must be addressed is the problem of network partitions and their effects on primary election. If a network partitions then this could lead to the situation where each replica group which is split across the partition has a different primary copy because it appears as though the original primary has failed to those replicas not in the same partition as the primary. The replication protocol to be presented can function correctly in the presence of network partitions.

6.5.1: The Algorithm.

In any passive replication scheme there must exist a means of electing a new primary should the existing one fail. A static ranking can be used: the order in which the replica group members appear in the Group–View Database will determine the order in which they are elected as primary copies should the current primary fail. Cohorts monitor the state of the primary by periodically sending "are you alive?" messages to it. The Group–View Database and the Exclude List are again used to impose a consistent view of a replica group on both clients and servers. As in the previous protocol, the database is

used to ensure consistency in the presence of network partitions. Initially we shall assume that the database is not replicated, but we shall show how it is possible to replicated it using this replication protocol.

This protocol does not require a reliable group RPC mechanism to function correctly, and as such we shall assume that the group RPC mechanism does not guarantee reliable delivery of messages.

6.5.1.1: Operation Invocations.

A client wishing to make use of a replicated service first obtains the Available List for the service from the database. The client then calls *initiate* to active the replicas mentioned in the Available List. Included in this *initiate* request will be the identity of the current primary, which the client received from the Group–View Database. Only the primary replica will execute requests, the other replicas will simply be placed into a monitoring mode in which they wait for one of two events: the primary sending them a message indicating the request has been completed, or the primary is seen to have failed (it does not respond to an "are you alive?" message).

Upon receiving the request the primary replica executes it and upon completion it broadcasts its new state to the passive replicas. They use this information to update their states to become consistent with the primary. When the backups have updated their states they send acknowledgments to the primary. The primary will wait to collect all of the replies, assembling an Exclude List for those backups which do not respond. The primary then contacts the Group–View Database and atomically updates the view of the group held there i.e., it removes from the view all backup replicas which did not respond. When the update is complete the primary sends the reply to the client.

The updating of the Group–View Database by the primary to reflect any changes in the group membership, and the actual updating of the states of the backups, must be done atomically (Section 6.3.2.5). The cohorts cannot commit their newly modified states until the groupview has been successfully committed to indicate which members have

up–to–date states. If this database operation fails then the cohorts must undo the latest checkpoint. This is necessary to ensure that in the event of a primary failure, only a backup which has a consistent state can be elected as the new primary.

Thus, each RPC is performed as an atomic action, within which the primary executes the client's request, checkpoints its state to its backups, updates the Group–View Database, and then replies to the client.

6.5.1.2: Cohort Failures.

All cohort failures are handled by the primary which is also responsible for maintaining the groupview. If a cohort fails to receive the initial request from the client but is available to receive the checkpoints from the primary then there is no problem with consistency as all cohorts which receive checkpoints will have the same state. If a backup fails to acknowledge a checkpoint message from a primary then the primary will exclude it from the groupview. This prevents it from being elected as a subsequent primary for this group until it has been brought up–to–date.

6.5.1.3: Primary Failure.

If a cohort detects that the primary has failed (either because the client has retransmitted the initial request or because the primary has not responded to an "are you alive" message) it must begin the primary election protocol. The cohort becomes the coordinator in this protocol and then contacts the Group–View Database to obtain the current groupview for this replica group. If this cohort is not mentioned in the groupview then the replica aborts the operation and will attempt to perform an update action later. This is possible if this coordinating replica has been previously excluded from the groupview but was not aware of this fact.

If the replica is in the current groupview then it will communicate with the replica which is listed as being the next primary in the group. If this replica also determines that the current primary has failed then it will assume the role of new primary, otherwise it will reply to the election coordinator with enough information to assure it that the primary is

active (this then stops the election coordinator from contacting the next replica in the groupview list). If this new primary has also failed then the protocol continues until a functioning replica is found to take over.

If a new primary is called for, then the replica which is elected must first atomically update the groupview held at the database to remove the old primary from it, and then broadcast an acknowledgement to the remaining group members (otherwise the remaining cohorts will determine that this replica has failed too). The new primary can then begin to execute requests.

6.5.1.4: Multiple Primaries.

From the above discussion it is still possible for multiple primaries to be in existence simultaneously. However, the Group–View Database prevents multiple primaries from being able to reply to the client. Because the client does not explicitly communicate with one replica in the group but simple broadcasts to the entire group membership, primary changes are not noticed by the client.

If a new primary is elected when the old primary is still in operation (though perhaps running on an overloaded node) then the new primary will complete the request. The old primary must contact its cohorts before it can communicate with the client and they will inform it that it has been overthrown as primary (and it can then execute recovery actions before rejoining the group). If the primary cannot communicate with the cohorts then it will assume they have failed and contact the Group–View Database to exclude them (again before it has communicated with the client) and find that it has been excluded from the groupview by the new primary.

6.5.1.5: Recovering Nodes.

Recovering nodes must perform update actions on those replicas which are out–of–date in the way described for the previous replication protocols. When these updated replicas are added to the group–view they can be placed back into their original position in the view list, even if this places them before the current primary.

6.5.1.6: Replica Updates.

Replicas which have been excluded from the groupview must run update actions before they can be allowed to rejoin and receive further checkpoints from the primary. This can be done in various ways: the checkpoint information sent by the primary is enough to update a replica's state, so the replica could join the group (although not the groupview at the database) and receive the next checkpoint from the primary before contacting the database to include itself in the groupview. Another mechanism would be to use an Update Daemon, as described in Section 6.4.4.

6.5.1.7: When to Checkpoint.

The frequency with which the primary replica checkpoints its state to its backups dictates how long it will take the primary to respond to a given client request, how much work must be re-executed to bring the new primary with the most recent checkpointed state to the same state that the failed primary had before it crashed, and therefore how many requests and replies clients will have to re-transmit and re-evaluate (as described in Section 3.3.2.5). The obvious solution is to checkpoint after every operation which causes the state to be modified. However, this can result in too great an overhead for certain applications, and so we believe that the checkpoint frequency should be related directly to the application and to the distributed system's characteristics.

Since all operations occur within an atomic action, we propose that the primary need not checkpoint its state until the commit phase of the client action. If the primary fails during the action then the client must abort and restart the action, receiving a new primary to service its requests. This method has the advantage of allowing all operations to proceed at the speed of the primary and not at the speed of the slowest backup.

When the action commits, the primary checkpoints its state to its backups and acknowledges the client. Failures of the primary after this mean that a new primary can be elected which is in a consistent state with the old primary when it committed. We have therefore made a distinction between availability of replicas during an atomic action, and

availability of replicas after an atomic action has terminated. For some applications this distinction can mean reducing the overhead of replication whilst at the same time providing availability where it is required.

6.5.1.8: Group–View Database Replication.

The replication of the Group–View Database is assumed to also be in a passive manner, with a primary database to which all requests are directed. However, the replication scheme for the database replica group is slightly different than for a more general server replica group.

Each database replica monitors the other replicas as well as the primary, maintaining a groupview for the Group–View Database (the order the replicas occur within the groupview is the order in which they will be elected as primary). The primary database replica checkpoints its state to its backups whenever a change to its state occurs. This is a two–phase protocol: in the first phase of the protocol the primary transmits the new checkpoint and possibly a new groupview for the database group. The backups acknowledge this, and if no further failures are detected the primary will inform the backups that they can commit their states. Otherwise the primary will abort this update action and start again with a new groupview.

Group–View Database replicas which fail to complete this protocol, either because they have crashed or become partitioned from the primary, must perform recovery actions as they could potentially possess out–of–date states. To update their states the database replicas must contact the current primary. Until recovery is complete these replicas will appear as though they have failed.

This two–phase protocol is also used by the primary whenever a database replica fails. If a primary finds that it can no longer communicate with a *majority* of the database replicas mentioned in its current groupview then it must stop executing as the primary may now be in a minority partition.

The election of a new database primary occurs in a similar manner to that described in Section 6.5.1.3. However, for a new primary to be elected, a *majority* of the database backups mentioned in the groupview must be available to participate. If this is not the case then a re-election cannot occur. This is to prevent two Group-View Database primaries from operating if a network partition occurs. With this protocol, at most one Group-View primary can be active in a distributed system, and therefore only those client and server replica groups which occur in the same partition as the database primary can also be active.

6.6: Summary.

This chapter contained detailed descriptions of a family of replication protocols which have been designed to operated on an object-oriented system which supports atomic actions. The system used to implement our replication protocols is Arjuna, and it was described in the opening sections.

We then described how our replication protocols operate to maintain replica consistency: using unordered multicasts at the communication level, and to impose ordering at the application level by relying upon the serialisability property of atomic actions. This has the desirable property that ordering is imposed only where strictly necessary. We then went on to describe the *Group-View Database* which maintains information about replica groups (the *groupview*) and individual replicas, and is used in all of the replication protocols described. The database is used to impose a consistent view of a given replica group on replicated clients, and is also used by recovering nodes so that they can run update operations on those replicas which are out-of-date as a result of the node failure. The concept of the *Exclude List* was introduced as a means of updating groupview information held at the database and as an efficient method of only updating those objects which need to be updated when they recover.

The first replication protocol was then described in detail, and it was shown how clients can make use of replica groups and how the replication protocol ensures that members of

a replica group remain consistent by making use of the Group–View Database and the reliable group communication layer. It was also shown how the concurrency control in Arjuna (strict two–phase locking in the initial implementation) can be extended to work on replica groups. It was shown how the commit protocol of the atomic action was modified so that replica group interaction can be taken into account. In this replication protocol, as long as a single replica remains operational at the end of an action then the action can commit. Finally, some the implementation of this replication protocol was described along with timings which have been taken.

The remaining sections of this chapter discussed how this first replication protocol can be modified to operate under an additional failure assumption: network partitioning. The two protocols described can both tolerate network partitions: the first one is an active replication protocol, whereas the second scheme is a passive replication protocol.

7: Conclusions.

This chapter will summarise the material which has been covered in this thesis and given an indication of the possible areas of future research.

7.1: Thesis Summary.

This first part of this thesis concentrated on the need for replication in distributed systems to aid in providing fault-tolerant applications. The advantages of using replication were outlined: increased availability of a replicated service, and increased performance of client and server interactions.

The next section of the thesis introduced the basic techniques used for the construction of reliable distributed applications. Atomic actions provide an integrated mechanism which addresses the problems of inconsistencies due to partial failures of an application, and interference between concurrent parts of an application. The object-oriented programming methodology was discussed as a useful way for structuring applications. Finally, the remote procedure call was described as a way in which remote objects can communicate, maintaining the concept of a local procedure call and so providing access transparency. Multicast communication can be used when multiple destinations are required to receive the same message, which introduced the notion of a *replica group*.

In the next section the principles behind object replication were described, starting with a classification of the various types of failures which can occur within a distributed system and which replication is typically used to mask. Then the two types of replication techniques were described: active replication, and passive replication. The differences between these two techniques were mentioned, along with the assumptions that they make about the distributed system on which they operate, and the requirements they impose on the underlying communication layer. It was shown that active replication (which is best modelled using the State Machine approach), is more difficult to provide because of its requirements, but is capable of tolerating larger classes of failures. To maintain

consistency between the members of an active replica group it is necessary that they all receive the same set of message in the same order. Finally, it was shown how the different replication classifications can be used to mask the failure types described.

The following section then continued the discussion about the communication layer. The various delivery properties which can be guaranteed by communication protocols were described, and it was shown that to maintain consistency between members of an active replica group we can use a communication protocol which ensures that all functioning members of the group will receive the same set of messages, without imposing an ordering on those messages. The ordering can be imposed at the application level by making use of the serialisability property of atomic actions. The communication protocol we used (rel/REL) was then described. It was then described how this protocol was used in the design and implementation of a reliable group RPC mechanism (GRPC). This GRPC was tested for various situations and the results were discussed. This section finished with a discussion of further modifications which must be made to the communications layer to ensure that replicas remain consistent despite events such as message buffer overflow and local timeouts, which cannot be solved by the delivery properties of the communication layer alone.

The next section described various replication protocols which have been used in database systems to replicate data. There was also a detailed description of some distributed systems which support either active or passive replication of objects and processes. The aim of this section was to show how each system provided replication and maintained consistency between replicas despite failures in the distributed system.

The final section described the design of three new replication protocols (which can be used in the Arjuna distributed system). The design of each protocol was discussed, and it was shown how by relaxing the assumptions which are made about the distributed environment and the communications layer, it is necessary to modify the replication protocols. All of the protocols described use atomic actions to impose concurrency control

and consistent ordering of operations where necessary. A Group–View Database and Exclude List are used to maintain replica consistency despite failures, and the Exclude List is used to ensure that recovering nodes can become available quickly by only having to execute update operations on those objects which are out–of–date. One of the replication protocol described (Available Objects) was implemented in Arjuna and an application was written which made use of the replication facilities provided. The results of this implementation were shown and discussed.

7.2: Main Contributions.

The contributions made by this thesis can be summarised as follows:

(i) it has been stated that in active replication the replicas need to receive the same set of message in the same order; this thesis has shown that such ordering can be imposed by atomic actions, enabling cheaper communication primitives to be used which ensure reliable delivery of messages. These ideas have been implemented in our system.

(ii) the discussion about replica consistency showed that although guaranteeing delivery of messages can solve some problems, problems such as message buffer overflow and local timeouts can still lead to replica state divergence. Two protocols have been described which solve these problems, leading to the design of a reliable group RPC mechanism.

(iii) this thesis has discussed and developed techniques for object replication. It has been shown how it is possible to distinguish the type of the operation being invoked on an object and how a replication protocol can take advantage of this information. The techniques proposed in this thesis for object replication are also transparent: users of replicated objects interact with only a single interface to the group, which hides the replication protocol.

(iv) a family of replication protocols have been developed for use in distributed systems. Each replication protocol makes various assumptions about the distributed

environment and the communications layer upon which it relies. We have shown how it is possible to modify the basic replication protocol (Available Objects) when the failure assumptions do not allow the use of reliable group communications primitives, and still provide flexible and efficient replication. The Available Objects replication protocol presented has also been implemented and results from this were presented.

7.3: Future Work.

Work into object replication has raised many interesting points, some of which have been briefly mentioned here, and which are currently being investigated as part of the continuing work in the Arjuna project. The work detailed in this thesis will be fully implemented as part of Arjuna.

- *Dynamically Maintaining Replication Levels*: it has been mentioned in previous chapters that replicas which join groups can do so only at specific times i.e., when no other action is currently using the group. This is to maintain consistency of the groupview between replicated clients. However, it would be better if replicas can join a group at any time, especially if they are replicas which have recovered after failing and so are needed to maintain a level of availability. To be able to do this is complex as consistency needs to be maintained between the replicated servers and any clients which are using them. Order preserving communications protocol can be used, but we shall be examining whether it is possible to maintain a level of replication (availability) without the use of such protocols. *Regeneration* of failed replicas is another area of future research. If it is possible to regenerate failed replicas quickly, then it is possible to maintain a level of availability during the lifetime of an application. Such regeneration of failed replicas requires the existence of a dynamic group membership protocol. The regeneration mechanism also provides the basis for an object-migration scheme which we are also examining: objects could be moved to the nodes on which their clients currently reside.

- *Object Placement*: although replication can be used to increase availability, if it is not used properly then availability can actually decrease e.g., if replicas have a source of common mode of failure. Further, the performance of an application can depend upon the number of replicas in use e.g., the more replicas in a group which have to be written to, the slower the response from a write operation. It is possible to maintain a level of availability for a replicated service with a smaller number of replicas if those replicas are located on reliable nodes. This can also improve the performance of an application because every client–replica group interaction need communicate with only a small number of replicas. Sometimes the choices involved in providing high availability conflict with high performance goals. There is a need for automating a method of finding the optimum location for replicas and the optimum number of replicas required for each application, given sufficient data about the distributed system and the requirements of the application e.g., the level of availability that is required and the performance.

- *User Level Group Management*: It has also become apparent from this work, and from building applications which make use of groups, that some way of allowing application programmers to manipulate groups would be useful. Currently, groups are a low level structuring technique e.g., for replica groups, and application programmers have no direct access to them. There are several situations in which it would be beneficial to application programmers if they could have direct access to groups, creating their own where necessary e.g., a programmer may wish to group several distinct objects together so that a message could be multicast to them all.

References.

[Abbadi 89]

A. El Abbadi and S. Toueg, "Maintaining Availability in Partitioned Replicated Databases", ACM Transactions on Database Systems, June 1989.

[Abbadi 90]

A. El Abbadi and D. Agrawal, "The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data", Proceedings of 16th International Conference on Very Large Databases, August 1990.

[Ahamad 87]

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.

[Alonso 90]

R. Alonso and L. L. Cova, "Managing Replicated Copies in Very Large Distributed Systems", Proceedings of the IEEE Workshop on Replicated Data, November 1990.

[Alsberg 76]

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.

[Anderson 81]

T. Anderson and P. A. Lee, "Fault Tolerance: Principles and Practice", Prentice-Hill, 1981.

[ANSA 89]

"Advanced Networked Systems Architecture (ANSA) Reference Manual", Volume A, Release 1.00, Part VI, Computational Projection, March 1989.

[ANSA 90]

"A Model for Interface Groups", ANSA, ISA Project, APM/RC.093.00, May 1990.

[ANSA 91a]

"An Abstract Model for Groups", ANSA, ISA Project, APM/RC.259.01, June 1991.

[Banatre 86a]

J.P. Banatre, M. Banatre, and F. Ployette, "An Overview of the Gothic Distributed Operating System", INRIA-Rennes Research Report No. 504, March 1986.

[Banatre 86b]

J.P. Banatre, M. Banatre, and F. Ployette, "The Concept of Multifunction, a General Structuring Tool for Distributed Operating System Structuring", Proceedings of the 6th Distributed Computing Systems Conference, Cambridge, MA, May 1986.

[Barrett 90]

P. A. Barrett et al, "The Delta-4 Extra Performance Architecture (XPA)", Proceedings of FTCS-20, Newcastle upon Tyne, June 1990.

[Bernstein 87]

P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.

[Birman 85]

K. Birman et al, "Implementing Fault-Tolerant Distributed Objects", IEEE Transactions on Software Engineering, June 1985.

[Birman 87a]

K. P. Birman and T. A. Joseph, "Reliable Communication in the Presence of Failures", ACM Transactions on Computer Systems, February 1987.

[Birman 87b]

K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems", in 11th Symposium on Operating System Principles, ACM SIGOPS, November 1987.

[Birman 88]

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.

[Birrell 84]

A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, February 1984.

[Birwhistle 73]

G. M. Birwhistle, O-J. Dahl, B. Myhrhaug and K. Nygaard, "Simula Begin", Academic Press, 1973.

[Black 86]

A. Black, et al, "Object Structure in the Emerald System", Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, October 1986.

[Chang 84]

J. Change and N. F. Maxemchuk, "Reliable Broadcast Protocols", ACM Transactions on Computer Systems, August 1984.

[Cheriton 84]

D. R. Cheriton and W. Zwaenepoel, "One-to-Many Interprocess Communication in the V-System", Department of Computer Science, Stanford University Report No. STAN-CS-84-1011.

[Cheriton 85]

D. R. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V Kernel", ACM Transactions on Computer Systems, May 1985.

[Cooper 84a]

E. C. Cooper, "Replicated Procedure Call", Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing, August 1984.

[Cooper 84b]

E. C. Cooper, "Circus: A Replicated Procedure Call Facility", Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems, October 1984.

[Cooper 85]

E. C. Cooper, "Replicated Distributed Programs", SOSP 10, December 1985.

[Cristian 85]

F. Cristian et al, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement", 15th International Conference on Fault-Tolerant Computing, Michigan, June 1985.

[Cristian 90]

F. Cristian, "Synchronous Atomic Broadcast For Redundant Broadcast Channels", IBM Research Report No. RJ 7203, Yorktown Heights, April 1990.

[Dahl 70]

O-J, Dahl, B. Myhrhaug and K. Nygaard, "SIMULA Common Base Language", Norwegian Computing Centre S-22, Oslo, Norway, 1970.

[Davcev 89]

D. Davcev, "A Dynamic Voting Scheme in Distributed Systems", IEEE Transactions on Saoftware Engineering, January 1989.

[Davidson 84]

S. B. Davidson, "Optimism and Consistency in Partitioned Distributed Database Systems", ACM Transactions on Database Systems, September 1984.

[Dixon 87]

G. N. Dixon, S. K. Shrivastava and G. D. Parrington, "Managing Persistent Objects in Arjuna: A System for Reliable Distributed Computing", Proceedings of the Workshop on Persistent Object Systems, Persistent Programming Research Report 44, Department of Computational Science, University of St. Andrews, August 1987.

[Dixon 89]

G. N. Dixon, G. D. Parrington, S. K. Shrivastava and S. M. Wheater, "The Treatment of Persistent Objects in Arjuna", in Proceedings of ECOOP 89. European Conference on Object Oriented Programming, University of Nottingham, July 1989. (also in The Computer Journal, Vol. 32, No. 4, April 1989)

[Downing 90]

A. R. Downing, I. B. Greenberg, and J. M. Peha, "OSCAR: A System for Weak-Consistency Replication", Proceedings of the IEEE Workshop on Replicated Data, November 1990.

[Eager 83]

D. L. Eager and K. C. Sevcik, "Achieving robustness in distributed database systems", ACM Transactions on Database Systems, September 1983.

[Eswaran 76]

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notion of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11, November 1976.

[Ezhilchelvan 91]

P. Ezhilchelvan, M. Little, and S. K. Shrivastava, "Implementing Reliable Group Invocations", Newcastle University Technical Report.

[Garcia-Molina 90]

H. Garcia-Molina and D. Barbará, "The Case for Controlled Inconsistency in Replicated Data", Proceedings of the IEEE Workshop on Replicated Data, November 1990.

[Gifford 79]

D. K. Gifford, "Weighted Voting for Replicated Data", 7th Symposium on Operating System Principles, December 1979.

[Goldberg 83]

A. Goldberg and D. Robson, "Smalltalk-80: The Language and its Implementation", Addison-Wesley, 1983.

[Gray 78]

J. N. Gray, "Notes on Data Base Operating Systems", in Operating Systems An Advanced Course, Lecture Notes in Computing Science, Vol. 60, Springer-Varlag 1978.

[Herlihy 85]

M. Herlihy, "Comparing How Atomicity Mechanisms Support Replication", Technical Report CMU-CS-85-123, Carnegie-Mellon University, May 1985.

[Hughes 86]

F. L. Hughes, "Multicast Communications in Distributed Systems", Ph.D Thesis, Computing Laboratory, University of Newcastle upon Tyne, October 1986.

[Jajodia 89]

S. Jajodia and D. Mutchler, "A Pessimistic Consistency Control Algorithm for Replicated Files which Achieves High Availability", IEEE Transactions on Software Engineering, January 1989.

[Jalote 89]

P. Jalote, "Resilient Objects in Broadcast Networks", IEEE Transactions on Saoftware Engineering, January 1989.

[Kernighan 78]

B. W. Kernighan and D. M. Ritchie, "The C Programming Language", Prentice–Hall, Englewood Cliffs, New Jersey, 1979.

[Ladin 90]

R. Ladin, B. Liskov, and L. Shrira, "Lazy Replication: Exploiting the Semantics of Distributed Services", Proceedings of the 9th ACM Symposium on Principles of Distributed Computing, August 1990.

[Lamport 78]

L. Lamport, "Time, clocks, and the ordering of events in a distributed system", Communications of the ACM, Vol. 21, July 1978.

[Lamport 82]

L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", ACM Transactions on Programming Languages and Systems, July 1982.

[Laprie 90]

J. C. Laprie, "Dependability: Basic Concepts and Associated Terminology", IFIP WG 10.4

[LeBlanc 85]

R. J. LeBlanc and C. T. Wilkes, "Systems Programming with Objects and Actions", IEEE 1985

[Liang 90]

L. Liang, S. T. Chanson, and G. W. Neufeld, "Process Groups and Group Communications: Classifications and Requirements", IEEE Computer, February 1990.

[Lippman 89]

S. B. Lippman, "C++ Primer", Addison–Wesley, 1989.

[Liskov 79]

B. Liskov, R. Atkinson, T. Bloom, J. E. B. Moss, C. Schaffert, R. Scheifler and A. Snyder, "Clu Reference Manual", Technical Report MIT/LCS/TR-225, MIT Laboratory for Computer Science, Cambridge, Mass., October 1979.

[Liskov 87a]

B. Liskov, "Implementation of Argus", Proceedings of the 11th ACM Symposium on Operating Systems Principles, November 1987.

[Liskov 87b]

B. Liskov and L. Shrira, "Promises: an Efficient Procedure Call Mechanism for Distributed Systems", Programming Methodology Group Memo, Laboratory for Computer Science, MIT, 1987.

[Liskov 88]

B. Liskov, "Distributed Programming in Argus", Communications of the CACM, March 1988.

[Lomet 77]

D. B. Lomet, "Process structure, synchronisation and recovery using atomic actions", in Proceedings of ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol. 12, No. 3, March 1977.

[Meyer 88]

B. Meyer, "Object-oriented software construction", Prentice-Hall, 1988.

[Mishra 89]

S. Mishra, L. L. Peterson, and R. D. Schlichting, "Implementing Fault-Tolerant Replicated Objects Using Psync", TR 89-3, Department of Computer Science, University of Arizona, Tucson, April 1989.

[Moss 81]

J. E. B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", Technical Report MIT/LCS/TR-260, Massachusetts Institute of Technology Laboratory for Computer Science, April 1981.

[Nett 85]

E. Nett et al, "Profemo: Design and Implementation of a Fault Tolerant Distributed System Architecture", GMD-Studien, No. 100, Technical Report, GMD, St. Augustin, June, 1985.

[Ng 89]

T. P. Ng and S. S. B. Shi, "Replicated Transactions", The 9th International, Conference on Distributed Computing Systems, Newport Beach, CA, June 1989.

[Noe 86]

J. D. Noe and A. Andreassian, "Effectiveness of Replication in Distributed Computer Networks", Department of Computer Science, University of Washington, Technical Report No. 86-06-05.

[Oki 88]

B. M. Oki, "Viewstamped Replication For Highly Available Distributed Systems", PhD Thesis, MIT Laboratory for Computer Science, August 1988.

[Olsen 91]

M. H. Olsen, E. Oskiewicz, and J. P. Warne, "A Model for Interface Groups", Proceedings of SRDS-10, Pisa, October 1991.

[Panzieri 88]

F. Panzieri and S. K. Shrivastava, "Rajdoot : a remote procedure call mechanism supporting orphan detection and killing", in IEEE Transactions on Software Engineering, Vol. SE-14, No. 1, January 1988.

[Pâris 86]

J.-F. Pâris, "Voting with Witnesses: A Consistency Scheme for Replicated Files", Proceedings of the 6th International Conference on Distributed Computing Systems, May 1986.

[Parrington 88a]

G. D. Parrington, "Management of Concurrency in a Reliable Object-Oriented Computing System", Ph.D Thesis, Technical Report TR/277, Computing Laboratory, University of Newcastle upon Tyne, December 1988.

[Parrington 88b]

G. D. Parrington and S. K. Shrivastava, "Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems", in Proceedings of ECOOP 88. European Conference on Object Oriented Programming, Norway, August 1988.

[Parrington 89]

G. D. Parrington, "Distributed Programming in C++ via Stub Generation", in preparation, Computing Laboratory, University of Newcastle upon Tyne, 1989.

[Peterson 87]

L. L. Peterson, "Preserving Context Information in an IPC Abstraction", Proceedings of the Sixth Symposium on Reliability in Distributed Software and Database Systems, Williamsburg, March 1987.

[Pu 86]

C. Pu, J. D. Noe, and A. Proudfoot, "Regeneration of Replicated Objects: A Technique and Its Eden Implementation", 2nd International Conference on Data Engineering, Los Angeles, February 1986.

[Satyanarayanan 90]

M. Satyanarayanan and E. H. Siegel, "Parallel Communication in a Large Distributed Environment", IEEE Transactions on Computers, March 1990.

[Schaffert 86]

C. Schaffert, T. Cooper, B. Bullis, M. Kilian and C. Wilpolt, "An Introduction to Trellis/Owl", in OOPSLA '86 Conference Proceedings, September 1986.

[Schlichting 83]

R. D. Schlichting and F. B. Schneider, "Fail-Stop processors: An approach to designing fault-tolerant computing systems", ACM Transactions on Computer Systems, August 1983.

[Schneider 84]

F. B. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors", ACM Transactions on Computer Systems, May 1984.

[Schneider 90]

F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", ACM Computing Surveys, December 1990.

[Shrivastava 88]

S. K. Shrivastava, G. N. Dixon, F. Hedayati, G. D. Parrington and S. M. Wheater, "A Technical Overview of Arjuna: A System for Reliable Distributed Computing", Proceedings of UK IT 88 Conference, July 1988.

[Shrivastava 90a]

S. K. Shrivastava and P. Ezhilchelvan, "rel/REL: A Family of Reliable Multicast Protocols for Distributed Systems", Newcastle University Technical Report.

[Shrivastava 90b]

S. K. Shrivastava, P. Ezhilchelvan, and M. Little, "Understanding Component Failures and Replication in Distributed Systems", ISA Project Report:UNT/TR1.

[Shrivastava 90c]

S. K. Shrivastava et al, "Fail-Controlled Processor Architectures for Distributed Sysytems", Technical Report, Computing Laboratory, University of Newcastle upon Tyne, 1990.

[Shrivastava 91]

S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System", IEEE Software, January 1991.

[Schwarz 84]

P. M. Schwarz and A. Z. Spector, "Synchronizing Shared Abstract Types", in ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984.

[Stroustrup 86]

B. Stroustrup, "The C++ Programming Language", Addison Wesley, 1986.

[Tanenbaum 88]

A. S. Tanenbaum and R. van Renesse, "Voting With Ghosts", Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, CA, June 1988.

[Verissimo 89]

P. Verissimo, L. Rodrigues, and M. Baptista, "AMp: A Highly Parallel Atomic Multicast Protocol", ACM SIGCOMM'89, Austin Texas, September 1989.