Objects and Actions in Reliable Distributed Systems[1]

S.K. Shrivastava, G.N. Dixon, and G.D. Parrington

Computing Laboratory, University of Newcastle upon Tyne

Abstract

This paper describes a method for constructing robust distributed programs. The method is based upon the provision of *atomic actions* that operate upon *objects* (instances of abstract data types). We begin by constructing robust non−distributed programs using the atomic action mechanism and then proceed to show how robust distributed programs can be constructed in a similar fashion. Finally we briefly examine other prototype distributed systems and examine their approach to the reliability problem.

Keywords:

Fault Tolerance

Object Based Programming

Distributed Systems

1. This paper is a revised and extended version of Chapter 6, "Robust Distributed Programs" taken from the book "Resilient Computing Systems", ed. T. Anderson, Collins, 1985.

1. Introduction

Distributed systems pose reliability problems not frequently encountered in more traditional 'centralised' systems. A distributed system consisting of a number of computers (nodes) connected by some communication network is subject to independent failure modes of its components, such as nodes, links and operating systems. The lack of any centralised control can mean that part of the system can fail with other parts still functioning, thus leading to the possibility of abnormal behaviour of application programs in execution. We will consider a distributed system where the nodes provide various services (e.g. data storage, printing) which can be invoked by an application program. Such a program will be termed a *distributed program* whose execution will require a group of cooperating processes distributed over the system. It is natural to require that such a group of processes behave 'consistently' in the presence of failures. A very simple consistency requirement, expressed in terms of the behaviour of a distributed program, is that of *failure atomicity*: the program either terminates normally, producing the intended results, or is 'aborted', producing no results at all.

Ideally we would like programs to terminate normally in the presence of an arbitrarily large number of failures, but this may require so much redundancy in the system as to make it economically unattractive. Assume that a distributed program is subject to two kinds of failures: (i) transmission failures such as lost messages (a message sent by a process does not reach its destination); and (ii) node crashes (a node stops functioning). We can then state the following range of reliability requirements for a distributed program.

(1) A program will terminate normally despite the occurrence of a (fixed) finite number of transmission failures (but node crashes cause abortion as stated earlier).

(2) A program will terminate normally despite the occurrence of a (fixed) finite number of transmission failures and node crashes.

In either case, if the limitation on the number of transmission failures or node crashes is exceeded, the program is aborted. The two types of requirements listed above represent a range from fairly to most reliable. Indeed, a very appealing technique for constructing type *(2)* programs is to build them out of type *(1)* programs. In this paper we will concentrate mainly on programs of type *(1)*, which can be regarded as the basic building blocks of a software system, and then briefly discuss how the other type of program can be constructed.

So far we have discussed tolerance only to specified types of faults. If tolerance to unforeseen faults, such as software design faults, is required, then type *(1)* or type *(2)* programs can be used, say, as alternates in a *recovery block* [Horning et al. 74]. A recovery block is a special construct consisting of an *acceptance test* and a series of *alternates*. A computation executes the first alternate and checks the result by evaluating the acceptance test. If the test fails, the next alternate is executed, after performing state restoration, and the acceptance test evaluated again. A discussion on design fault tolerance is beyond the scope of this paper, but the interested reader is referred to [Lee and Anderson 85] for more details.

Having indicated the scope of the problem to be solved, let us investigate some structural properties of a distributed program. Most modern programming languages (e.g. C++ [Stroustrup 86]) support the facility of data abstraction, enabling a programmer to associate a set of operations with data structures. The term *abstract objects* (or objects, for short) will be used to refer to instances of data abstractions. Abstract objects are structured entities that can only be manipulated by invoking the procedures associated with them. Programs constructed using abstract objects have a hierarchical structure in that higher level objects are constructed using

lower level objects: making such a program reliable essentially involves making objects reliable (that is ensuring that objects remain consistent in the presence of failures of certain kinds). It is natural to assume that distributed programs will have a similar structure to that discussed above, the only difference being that objects may be distributed over the network. All that is then necessary is to provide network protocols enabling a program to invoke operations of a remote object. Such a protocol should provide the abstraction of invoking a procedure. Hence the term *remote procedure call* (RPC) is often used to refer to the services provided by the protocol. Our task is thus to investigate reliability techniques for constructing reliable distributed programs of type *(1)* and *(2)*, which in turn implies investigating reliability techniques for robust objects whose operations can be invoked by making use of RPCs. We begin by investigating the notion of atomic actions.

2. Atomic Actions

We will assume that objects can be shared between various programs. It is then necessary to ensure that concurrent executions of programs be free from interference, i.e. concurrent executions should be equivalent to some serial order of execution [Eswaran et al. 76, Best and Randell 81]. To understand this, consider the following two programs (where *w*, *x*, *y* and *z* are distinct variables):

P_1: $z := 10$; $x := x+1$; $y := y+1$

P_2: $w := 7$: $x :=$ x * 2; $y := y * 2$

Assume that $x=y=2$ initially. Then, a serial execution order '$P_1;P_2$' will produce the result $z=10$, $w=7$, $x=y=6$, and execution '$P_2;P_1$' will produce the results $z=10$, $w=7$, $x=y=5$. The partly concurrent execution order given below

($z := 10 \parallel w := 7$); $x := x+1$; $y := y+1$; $x :=$ x * 2; $y := y * 2$

will be termed *interference free* or *serialisable*, since it is equivalent to the serial order '$P_1;P_2$'. However, an execution order such as

($z := 10 \parallel w := 7$); $x := x+1$; $x := x * 2$; $y := y * 2$; $y := y+1$

is not free from interference since it cannot be shown to be equivalent to any serial order. Programs that possess the above mentioned 'serialisable' property are said to be *atomic with respect to concurrency* (or simply atomic), and the computations that are invoked by these programs are termed *atomic actions*. We will embellish such programs with the failure atomicity property mentioned earlier, thus ensuring that any computation progresses without interference from other computations and at the same time either terminates normally (producing intended results) or is aborted (terminated without producing any results). It is reasonable to assume that once a computation terminates normally, the results produced are not destroyed by subsequent failures (node crashes). Hence, we will also require that an atomic action be enriched with the *permanence of effect* property (i.e. the state changes produced should be *stable* or *durable*, with a high probability of surviving node crashes). An atomic action with the failure atomicity and permanence of effect properties will be termed a *robust atomic action*. In view of the discussion presented in the preceding section, we will investigate in subsequent sections the fault tolerance techniques that should be employed to support robust atomic actions and objects. We will begin this investigation by first considering local (non–distributed) computations and then discuss what extensions, if any, are required to support distributed computations.

We conclude this section by briefly dwelling on the topic of concurrency control for atomic actions. This subject has been, and continues to be, intensely researched [Bernstein and

Goodman 81], and although the research has been largely confined to database systems, the techniques proposed are applicable to other areas. A very simple and widely used approach is to regard all operations on objects to be of type 'read' or 'write', which must follow the well known synchronisation rule permitting 'concurrent reads but exclusive writes'. This rule is imposed by requiring that any computation intending to perform an operation that is of type read (write) on an object, must first acquire a 'read lock' ('write lock') associated with that object. A read lock on an object can be held concurrently by many computations provided no computation is holding a write lock on that object. A write lock on an object, on the other hand, can only be held by a computation provided no other computation is holding a read or a write lock. In a classic paper [Eswaran et al. 76], Eswaran *et al.* proved that all computations must follow a 'two–phase' locking policy (see Fig. 1) to ensure the atomicity property (i.e. lack of interference).

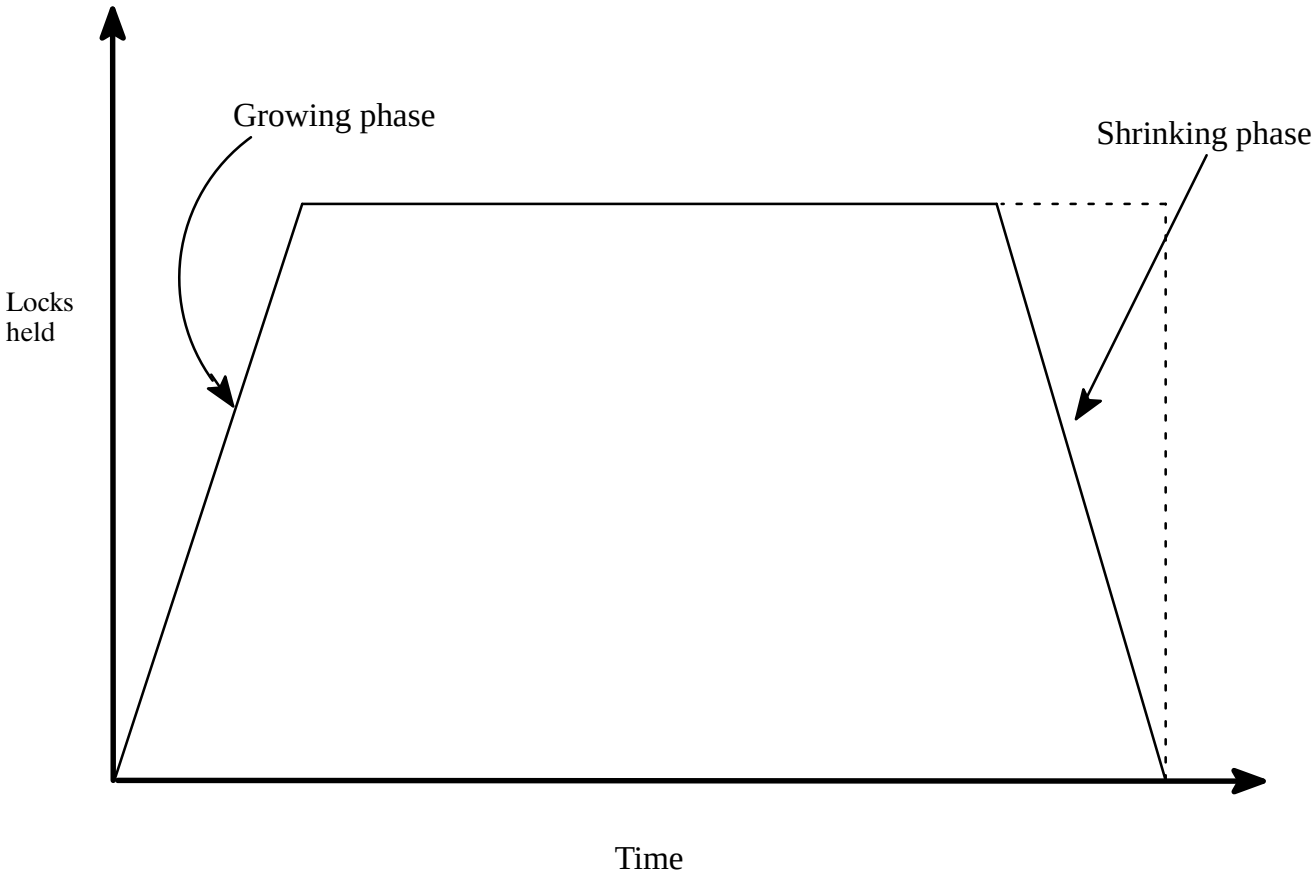


Figure 1: Two – phase locking.

During the first phase, termed the growing phase, a computation can acquire locks, but not release them. The tail end of the computation constitutes the shrinking phase, during which time held locks can be released but no locks can be acquired. Now suppose that a computation in its shrinking phase is to be aborted, and that some objects with write locks have already been released. If some of these objects have been locked by other computations, then abortion of the computation will require these computations to be aborted as well. To avoid this *cascade roll back* problem, it is necessary to make the shrinking phase 'instantaneous', as shown by the dotted lines. In effect this means that all the held locks are released simultaneously. For the sake of simplicity, in the rest of the paper we will assume that some concurrency control technique, such as two–phase locking, is being employed and, further, that there is no danger of cascade roll back.

3. Fault Models and Exception Handling

We assume that the communication system connecting all the nodes is faulty, as are the nodes themselves. Communication system faults are modelled as being responsible for the following types of failures;

(i) a message sent by a node does not reach its destination;
(ii) a message can get corrupted during its passage;
(iii) messages can arrive out of order at a destination;
(iv) a message can get replicated during its passage.

We will assume that there is sufficient redundancy in a message (e.g. a checksum) to enable a receiver to detect corrupted messages with a very high probability of success. Thus, we need only concern ourselves with failures of types (i), (iii) and (iv). It will be the responsibility of the underlying RPC protocol to cope with these failures and to signal exceptions when failures cannot be masked (see section 5.2 for more on Remote Procedure Calls).

We will model node faults in a very simple manner. We assume that a node either works perfectly or it simply stops working (crashes). After a crash, the node is repaired within a finite amount of time and made active again. A node can have two types of storage facilities: stable (crash proof) storage, and non–stable (volatile) storage. All the data stored on non–stable storage is lost when a crash happens. All the data stored on stable storage, on the other hand, remains unaffected by a crash.

The reader may perhaps find our model of a faulty node less realistic than that of the communication system. The problem is that a malfunctioning node can be capable of arbitrary behaviour (and not just remaining silent as assumed here). Designing systems under an arbitrary failure mode assumption is an extremely difficult task and is not within the scope of this paper (but see [Lamport et al. 82] for further reading). If we assume that a malfunctioning node is quickly 'shut down' by some external agency, then our model can be said to approximate reality. Alternatively, or in addition, it is also possible to introduce enough redundancy in a node for its behaviour to approximate more closely to that of the model.

We will adopt the exception handling strategy proposed in [Cristian 82]. We will say that an invocation of a procedure either completes successfully (a normal return is obtained) or abnormally (a specified exceptional return is obtained). Let C be a procedure that can **signal** two exceptions e_1 and e_2; then, we will adopt the following notation to indicate this fact:

procedure C(...) **signals** e_1, e_2;
begin
    ...
    assert B<fail: ... ; **signal** e_1>;
    ...
    D(...)<p_1: ... ; **signal** e_2>;
    ...
end<fail: ... ; **signal** fail>;

The body of the procedure also illustrates several further points: (i) angular brackets enclose exception handlers; (ii) a default 'fail' exception is generated if the Boolean expression of the assert command evaluates to false; (iii) the handler for exception p_1 causes C to signal e_2; (iv) every procedure can signal a fail exception. The invoker of C can provide specific handlers for e_1, e_2 and fail:

C(...)<e_1:H_1 | e_2: H_2 | fail: H_3>;

where vertical bars separate handlers. The handler for a fail exception is intended to cope with the fail exception and all other exceptions generated for which no specific handler is available:

C(...)<fail: H>;

In this case H will 'catch' e_1 or e_2 or a fail exception.

4. Robust Non–Distributed Programs

4.1 Recovery Semantics

In this subsection we will present principles of backward error recovery. For the sake of simplicity, only local computations (non–distributed) will be considered. Using the terminology and concepts developed for an earlier paper [Anderson et al. 78], for a computation or a process to have the abstraction of backward error recovery (henceforth termed *recovery*) requires the provision of the following three primitives:

(1) *Establish a recovery point*. A process can establish a recovery point (*erp*), thus indicating the start of a new recovery region (see Fig. 2). This implies that, if necessary, the states of any recoverable objects modified in that region can be restored automatically to those at the start of the region.

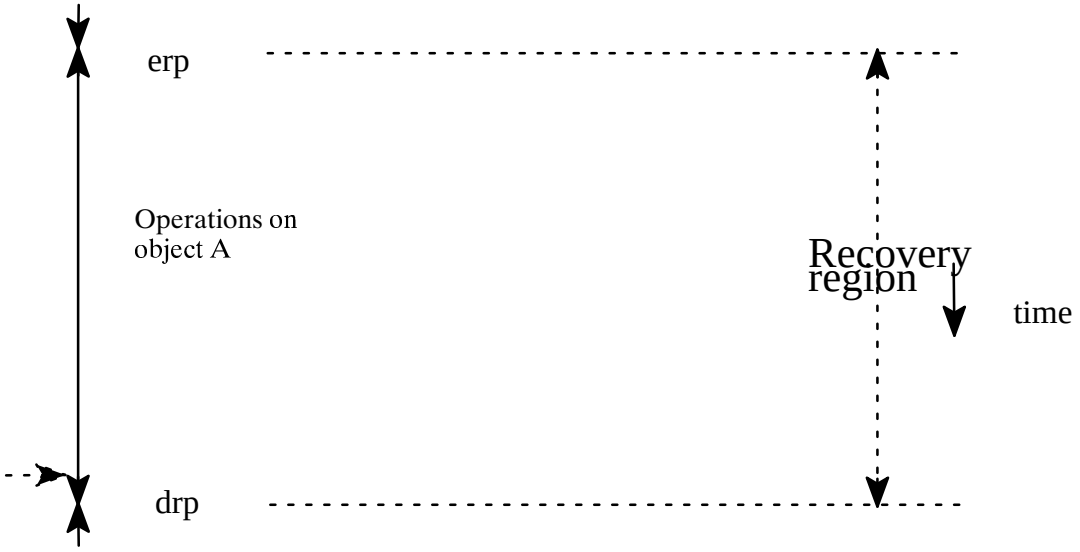


Figure 2: A recovery region.

(2) *Discard a recovery point.* A process can discard a previously established recovery point by invoking the operation *drp,* thus indicating the end of a recovery region.

(3) *Restore*. This operation invokes (backward) recovery.

By definition, a *recoverable object* is an object with the property that if its operations are invoked from within a recovery region, then the invocation of recovery will see the object automatically restored to its abstract state prevailing at the start of the recovery region. Of course, this also means that if a recoverable object is modified from outside a recovery region, then no automatic recovery capability will be available.

Let A be a recoverable object. From Fig. 2 we note that a number of operations have been performed on A. Suppose the flow of control has reached the point shown by the dotted arrow when recovery is invoked. How is object A restored? There can be several ways recovery can be mechanised. Here are two examples:

(1) When the executing process enters a recovery region, the current state of the object is recorded on some data structure (a *checkpoint* is taken). Recovery then involves the substitution of the recorded state for the current state. (In general, a 'reverse' procedure is required to perform recovery [Shrivastava and Banâtre 78]).

(2) Enough information is recorded about the operations performed since the recovery region was entered so as to be able to sequentially 'undo', in reverse order, all these operations.

The recovery cache algorithm for recovery blocks employs an optimised version of the first approach. We will assume a similar approach in the following discussion (see the next subsection).

As an example of the usage of these primitives, consider the following program, where S is any command.

```
erp;
        start: S<fail: restore;  goto start>
drp;
```

What will happen if an exception is detected during the execution of S? (S is retried.)

It is possible for recovery regions to be nested, as exemplified by, say, nested recovery blocks, but for the sake of simplicity we shall not consider nested recovery regions further. It is important, however, to understand what nesting of recovery regions implies. Consider the following recovery block program:

R_1: **ensure** at_1 **by** A.op(...) **else–by** B.op(...) **else** fail;

where A and B are two abstract recoverable objects, and the notation 'A.op( )' indicates the procedure 'op' exported by object A. Suppose that the body of this procedure contains a recovery block:

procedure op(...)
begin
        R_2: **ensure** at_2 **by** S_1 **else–by** S_2 **else** fail;
end

We ask the question: is R_2 nested within R_1? Irrespective of what the answer is, the semantics of program R_1 remain the same: if the test at_1 fails, the states of objects A and B will remain unchanged. The mechanisation of recovery required to support the above semantics will, however, crucially depend on whether R_2 is regarded as nested within R_1. These issues are discussed at length in the paper already cited [Anderson et al. 78]. Here we will adopt the view that R_1 and R_2 represent recovery regions at two different levels of abstraction; hence they are not nested (are *disjoint*). The program given below, on the other hand, does contain nested recovery regions.

R_3: **ensure** at_1 **by**
ensure at_2 **by** S_1 **else–by** S_2 **else** fail
else–by...

4.2 Nested Atomic Actions and Robust Objects

Let us ignore for the moment any issues concerning recovery and consider actions that are atomic with respect to concurrency. We will use the notation '**action** S **end**' to indicate that the execution of S will be atomic. Let A and B be two abstract objects:

A_1: **action** ...; A.op(...); ... ; B.op(...); ... **end**

We will adopt the view that any action, such as A_1, is composed out of smaller actions (which are themselves composed out of yet smaller actions, and so on). This is illustrated in Fig. 3, which depicts the structure of the computation invoked by A_1, and indicates that the execution of program A_1 gives rise to a computation with nested atomic action structure. Note that in program A_1, the nesting of actions is *implicit*. If we permit concurrency within an action, then nesting of actions has to be indicated *explicitly*:

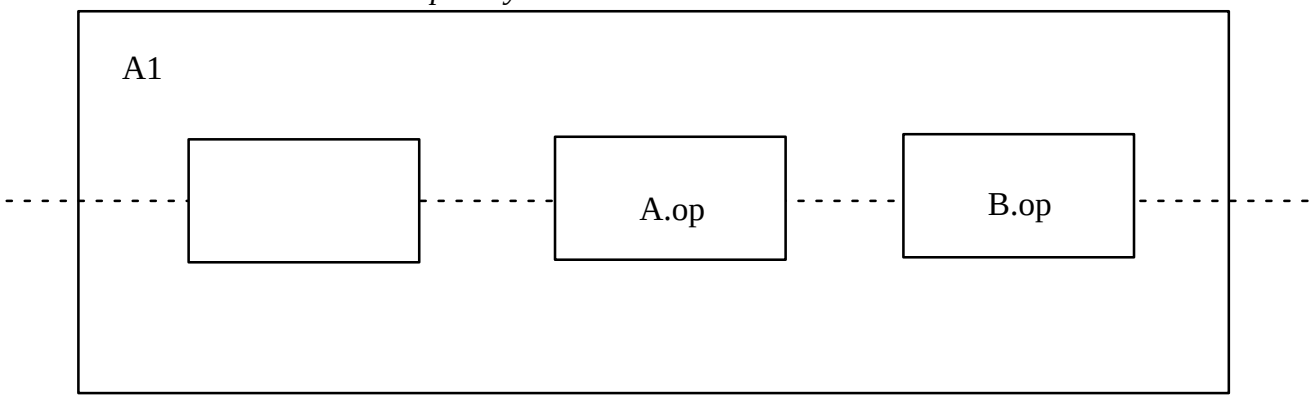


Figure 3: Nested atomic actions.

A_2: **action**
...
cobegin
A_3: **action** ... **end**
A_4: **action** ... **end**
coend
...
end

In the above program, concurrent actions (A_3 and A_4) are nested within A_2. In this paper we will primarily consider sequential programs. Nested concurrent actions provide a convenient means of constructing type *(2)* programs, as we shall see in section 5.4.

Any concurrency control technique employed must ensure that the execution of the entire program (e.g. A_1) is free from interference. So, if two–phase locking is being utilised, then all the locks (including those acquired by internal actions such as A.op(...)) must be held till the outermost action terminates, so as to avoid the danger of a cascade roll back [Moss 82].

Atomic actions provide a natural structure for introducing recoverability. We will now investigate the possibility of introducing failure atomicity and permanence of effect properties to

obtain robust actions. We will assume that outermost atomic actions are to be embellished in this manner. Let C_i be the stable (crash resistant) state of a system (i.e. C_i is the aggregation of stable states of objects). Then the function of a robust atomic action (with write operations) is to change the state C_i to some new stable state C_{i+1}. Assuming that '**r–action** S **end**' represents a robust action, we can visualise the implementation of 'r–action' and 'end' as depicted below (two–phase locking is assumed):

r–action	→	*erp*	→	record start of a recovery region;
end	→	*drp*	→	record new state on stable storage; release locks;

We assume the existence of a stable storage object that is used for storing state information of other objects. The exact details of how the recovery region data are managed need not concern us here but we will nevertheless briefly explain the basic principles involved.

To start with we will assume that every (recoverable) object is responsible for performing its own recovery. Assume that A_1 (Fig. 3) is a robust action and some process P is executing it. Then the table given below summarises the recovery aspects. We assume that a data structure, referred to as the *P–list*, is created for the executing process to record recovery data, which includes the names of recoverable objects accessed so far and information regarding the locks acquired.

Operation	*Actions of process P*
erp	create a data structure for recording recovery data; (call this 'P–list')
A.op()	if lock on A is not held then lock A; if name 'A' is not recorded in P–list then make an entry; call A.op( );
restore	for all object names recorded in P–list do: call objectname.recover;
drp	for all object names recorded in P–list do: call objectname.securestate; release locks on all objects whose names are in P–list; delete P–list.

We assume that every recoverable object provides a 'recover' procedure, the function of which is to adjust the internal variables of the object so that the abstract state of the object is restored to that at the start of the recovery region. In the simplest case, this will consist of reinitialising the objects to their latest stable states. Thus, whenever recovery is desired, process P executes 'restore' – this has the effect of restoring all the called objects to their stable states. Finally, when discarding the recovery point of A_1, we must ensure that a new stable state has been created by invoking the 'securestate' operation of all the objects recorded in the P–list so that each object can store its new state on stable storage. To summarise, a recoverable object has two special operations:

recover: restore the state of the object;

securestate: record the current object state on stable storage.

We can also add a third operation for coping with crashes; so crash recovery of a node involves calling the crash–recover operations of all the objects:

crash–recover: reinitialise the object to the current stable state.

Consider now the particular scenario depicted in Fig. 4, where A_1 is an outermost robust action. Suppose that objects A and B are recoverable. Assume that the operation currently in execution ($B.op_2$) terminates by signalling a fail exception and this calls for recovery of A_1. The backward recovery of A_1 will involve restoring objects A and B, and as we have seen, this is performed by automatically invoking the 'recover' operations of these two objects. So, a single call to 'A.recover' will undo state changes introduced by the two atomic actions ($A.op_1$, $A.op_2$). This illustrates a point not widely appreciated, and that is that recovery of an action (such as A_1) need not involve individually recovering constituent atomic actions. It is certainly possible to adopt a recovery model in which recovery of an action is performed by recovering constituent actions (e.g. [Moss 82, Liskov and Scheifler 83]), i.e. recovery is performed on a call by call basis. However, we consider that the recovery model presented here is a more elegant way of mechanising object based recovery. Similarly, in order to make the state changes produced by A_1 stable, it is not necessary that the state changes produced by each constituent action be made stable individually; rather, at the termination time all the recoverable objects used within A_1 are asked to make their current states stable (see the table of recovery actions above).

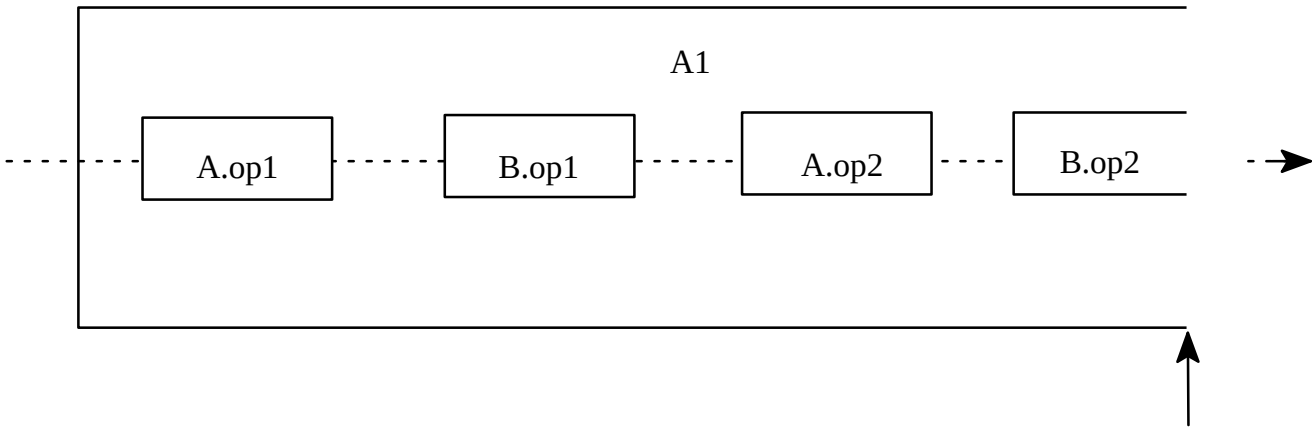


Figure 4: Nested atomic actions.

As a consequence of the above observations, internal actions have no need for stable storage. In particular, if an operation such as $B.op_1$ has recovery regions (the body of op_1, for example, uses recovery blocks) then upon termination, all the recovery data can be simply discarded. The details of managing recovery data at different levels of abstraction are somewhat involved and the reader is referred to [Shrivastava 81] for further details.

4.3 A Second Look At Crash Resistance

The framework for recovery presented above has one flaw which can be exposed by considering the following situation: a node crash occurs during the execution of the operation 'drp' of atomic action A_1. Suppose object A has been secured to the new stable state but the same is not true for object B. Thus, after recovering from the crash, while object A will be in the new state, object B will be in the original state – an inconsistency. What is necessary is for the crash recovery algorithm of the node to somehow or other either restore A to the prior stable state or to

bring B to the new stable state. Since the second option is not possible, we require that A be restored to the prior stable state. This suggests that the algorithm for 'drp' (for an outermost robust atomic action) be modified in such a manner that normal termination is performed only when *all* objects have made their states stable. The algorithm given below ensures this.

The commit algorithm

"implementation of drp"

```
        save process–state (cr); fate:=abort;
        [put 'process–state', 'fate' and 'P–list' on stable storage];

        for all object names in P–list do
              objectname. securestate;
        [fate:=commit];
cr:     case fate of
              abort: for all object names in P–list do
                                    {objectname.recover; release lock}
              commit:      for all object names in P–list do
                                    {objectname.commit; release lock}
        end "case"
        die "delete the process and the P–list"
```

Employing the terminology widely used, the termination of an action with the new state made stable will be referred to as *committing* the action, and the algorithm that determines whether an action is to be aborted or committed will be called the *commit algorithm*. The primitive drp (for an outermost robust action) implements the commit algorithm. We introduce a new operation for recoverable objects: 'commit' whose function is to make the most recent secured state the current state of the object. The operation 'securestate' is modified so that the object merely records the most recent object state on stable storage (which is not made the current state until the commit operation is executed). We assume that associated with the executing process, there is a variable named 'fate' which records the fate of the action (to be aborted or committed).

Looking at the algorithm, the process saves its state, ready to execute from label 'cr' and the variable 'fate' is initialised to the value 'abort'. A crash of the node at this point will result in the abortion of the action since all the objects will have been restored to the previous committed states, the recent states being discarded with locks released; at the same time, no state information about the process survives the crash. If there is no crash, the process executes the operations enclosed by brackets and makes itself crash proof (held locks also become stable). The execution of this set of operations itself must be failure atomic (this is indicated by the brackets), so that either all of the data regarding the process becomes stable or none is. Construction of stable storage which provides atomic update facilities is an interesting design exercise (see [Lampson and Sturgis 81] for details). The fate of the process is changed to commit only when all the objects have secured their states. If a crash occurs during this process (some of the objects have yet to make their states stable), then after crash recovery the crashed process will be recreated, ready to execute the commit algorithm from point 'cr'. As can be seen, in this case all the objects will be restored. The operations 'recover', 'commit' and 'release a lock' must be idempotent – multiple executions are the same as a single execution. (Do you see why? Consider what will happen if a crash occurs during the execution of the case statement. So, we must ensure that several executions of the case statement are equivalent to a single one.)

5. Robust Distributed Programs

Our next task is to extend the ideas developed in the previous section to distributed programs. We will do this in three stages. In the first subsection we will discuss a model of distributed computation, and follow this up with a discussion on issues concerned with design and use of remote procedure calls. In the last subsection we will discuss recoverability and commitment issues.

5.1 A Model of Distributed Computation

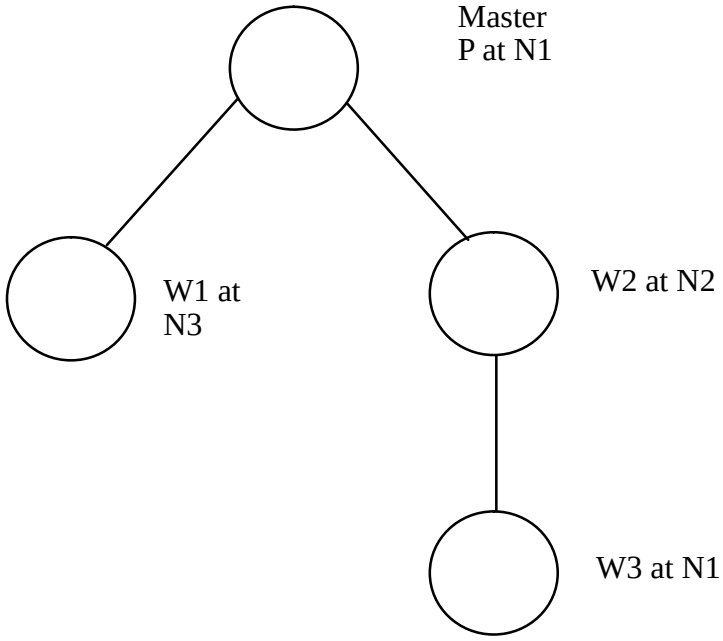
It was stated earlier that a distributed program will be executed by a group of processes. There can be several ways of structuring such a group. Here are three examples: (i) a new process is created to serve each remote call; (ii) each object manager has a few (internal) server processes that receive remote requests; (iii) a local process creates a server process on a remote node to execute all the calls directed at that node. We will adopt the last technique for our purposes, because it represents a very simple way of extending a non–distributed computation to a distributed one (the approach adopted in the Argus system [Liskov and Scheifler 83] roughly corresponds to the first case).

Consider the following example: a distributed program makes use of three objects A (on the local node N_1), B (on node N_2) and C (on node N_3); further, object B itself accesses some object D on node N_1. In Fig. 5, we have shown the program and the corresponding process structure. It is assumed that primitives 'rpc(workername, procedurename, ...)' and 'createworker(nodename, ...)' are available for invoking remote procedures and creating processes on remote nodes. The process executing this program creates a server process at a node before forwarding call requests to it. Thus process P will explicitly create processes W_1 and W_2. Object B is responsible for creating process W_3 at N_1 so that calls on object D can be made. We thus see that the execution of a distributed program can give rise to a hierarchy of processes.

This model of computation provides a straightforward means of incorporating the recovery and crash resistance properties discussed previously. So, for example, backward recovery involves each process calling the 'recover' operations of the objects it has directly manipulated: process P will invoke A.recover and also send 'restore' requests to processes W_1 and W_2. We will discuss these features in the section on Recovery and Crash Resistance.

```
createworker (N3,...);
”returns W1 as the name of
the created process”
rpc (W1, C.op1, ...);
createworker (N2, ...);
rpc (W2, B.op1, ...);
A.op1(...);
rpc (W2, B.op2, ...);
```

(a) A Distributed Program



(b) Process Structure

Figure 5: Distributed computation.

5.2 Remote Procedure Calls

In this subsection we will briefly examine remote procedure calls, whereby a *client* process (such as W_2, Fig. 5) can invoke an operation of a *server* or *worker* process (such as W_3, Fig. 5). Essentially, in this scheme a client's remote call is transformed into a request message to the named server who performs the requested work and sends the results back:

Client	*Worker*
	cycle
send() ”call msg”	receive() ”get work”
	”work”
receive()	send() ”send results”
	end

Both the client and server will contain measures for dealing with failures such as lost messages. The design and implementation of remote procedure calls continues to be a subject of ongoing research, and the reader’s attention is drawn to [Birrell and Nelson 84, Hamilton 84, Panzieri and Shrivastava]. RPC protocols, in common with other types of protocol, typically employ time–out mechanisms to prevent indefinite (or long) waiting by a process expecting a message. Suppose that a server crashes in the middle of a call, in which case the client will not receive the result message and will eventually be ’timed out’. It is interesting to note that a time–out exception could indicate any of the following four possibilities: (a) the server crashed during the call; (b) the server did not receive the request (the client’s call message got lost); (c) the server’s reply did not reach the client; (d) the server is still performing the work (so the time–out interval was not long enough). We will assume that the client’s time–out results in the call terminating abnormally with a fail exception. For the sake of simplicity, we will assume that this is the only exception a remote call can get.

rpc (servername, procname, params)<fail: handler>;

We will assume the following semantics for the RPC: if the call terminates normally (no fail exception is generated) then exactly one execution for the call has taken place at the server; on the other hand, if the fail exception is generated then it is not known whether the server has performed the work. This will be termed *exactly once* semantics. What actions should be taken by the handler associated with ’fail’? A reasonable strategy is to retry the call a few times:

rpc (...)<fail: retry(n)<fail: handler>>;

The above notation indicates that the call is retried a maximum of n times. If the call succeeds, then as before we require that only one execution has been performed at the server. If all the retries fail then one possibility, assuming the call has been made from within a recovery region, is to invoke backward recovery:

rpc (...)<fail: retry(n)<fail: restore>>;

The implementation of ’restore’ in a distributed environment will be discussed in the next subsection. It is also possible to select a ’stronger’ semantics for RPCs, termed *at most once*: if a call fails then no side effects will have been produced. This requires the capability of undoing side effects of individual calls; not surprisingly, this call semantics has been chosen by the advocates of the ’call by call’ recovery model [Liskov and Scheifler 83, Svobodova 84]. As can be appreciated, one advantage of this approach is that, when a call fails, it is not necessary to recover the enclosing action (or more precisely, to the nearest recovery point) as is the case with the model adopted here.

With the exception handling strategy just mentioned in mind, a simple technique exists for creating a worker at a remote node:

```
procedure createworker (Ni, ... )
begin
        ...
        rpc (manager at Ni, create, params)
                <fail: retry (n)<fail: signal fail>>;
end
```

This assumes that every node has a manager process that accepts calls for worker creation. The created worker enters the cycle for receiving calls (see the simplified RPC protocol at the beginning of this subsection) and sets a time out (the idle period). If the client's createworker call fails and a worker nevertheless is created, then this worker will never receive a request for work. hence its time out will expire and the associated handler can simply kill the worker. (Modification to this strategy is required if the worker has become stable – as we shall see later.)

5.3 Recovery and Crash Resistance

We will now extend the recovery model of the previous section so that: (i) if a process within the hierarchy of processes executes a 'restore' operation, then it and its siblings all perform backward recovery; (ii) commitment by the master process involves commitment by its siblings (which in turn will involve commitment of their siblings and so forth). We will consider all the five operations (erp, local call, rpc, restore, drp) and see what recovery management is implied.

(1) *Operation erp.* The executing process creates a data structure (P–list) to record recovery data.

(2) *Local call to an object.* If the object name is not on the P–list, then make an entry before the call.

(3) *Remote call.* If the name of the worker process is not on the P–list, then make an entry before making the call. The worker process will execute the call as a local call, i.e. the name of the object will be recorded in the P–list of the worker.

(4) *Operation restore.* The process performs the following operations: calls objectname.recover operations of all the local objects recorded in the P–list; sends restore commands to all the workers whose names are in the P–list. If a remote call fails then the process destroys itself (the outermost atomic action will eventually be aborted).

(5) *Operation drp.* The commit algorithm (for the outermost atomic action) to be executed by the master process, such as that shown in Fig. 5 is modified as shown below:

The commit algorithm of the master process

"implementation of drp"

```
save process–state(cr); fate:=abort;

[put 'process–state', 'fate' and 'P–list' on stable storage];

for all local object in P–list do
        objectname.securestate;
for all process names in P–list do
        rpc (processname, stable, . . .)<fail: retry(n)
                                        <fail: for all local object
```

```
                                        names in P–list do
                                        {objectname.recover;
                                         release lock}; die>>;
    [fate:=commit];
    cr: case fate of
        abort:   for all local object names in P–list do
                    {objectname.recover; release locks}
        commit: for all local object names in P–list do
                    {objectname.commit; release lock}
                 for all process names in P–list do
                    rpc(. . .,commit,. . .)
                    ”retry till success”
        end ”case”
    die;
```

Comparing with the original commit algorithm, we notice a few differences. The process secures local objects first then sends a 'stable' command to all the workers. This is a signal for a worker to make itself and its objects stable. If a remote call fails (after a few retries) then the master process performs local recovery and dies. We thus make it the responsibility of the workers to determine when to abort. A skeleton algorithm for a worker – modified from that given at the beginning of the section on Remote Procedure Calls – is given below:

Life of a worker

```
fate:=abort; create an empty P–list;

cycle
    cr: get–work(. . .) ”receive an rpc request with an idle time out”
        <fail: case fate of
                abort:  {restore local objects; die;}
                heldup:  checkifmasterup(. . .);
                          if up then goto cr else
                             {restore local objects; die}
            end ”case fate”>
    analyse–request;
    case command of
        restore:    ”see the description of command restore at the beginning of this
                sub–section”
        stable: save process–state(cr)
                [put 'process–state', 'fate', 'P–list' on stable storage];
                for all local object names in P–list do
                        objectname.securestate;
                for all process names in P–list do
                        rpc (processname, stable,. . .)
                            <fail: retry(n) <fail:restore local
                                    objects; die>>;
                result:=done;
                [fate:=heldup]
        call:  if lock on the named object not held then lock
                        the object;
               if name not in P–list then make an
```

```
                    entry; execute the call; prepare results;
            commit: "similar to the master"; result:=done;
        end "case command";
        send–results (. . .);
    end "cycle"
```

A worker waits for requests, with an 'idle time–out'. If the time out expires (a fail exception is generated) and fate=abort, then the worker aborts, suspecting a crash of its master. On the other hand, if the fate is 'heldup' then the worker cannot abort unilaterally, and must check if its master has aborted. For this purpose, we assume the existence of a procedure 'checkifmasterup (...)' which returns **true** if the called node has the named process running and **false** otherwise. Looking at Fig. 5, when process P executes 'drp', the operation will terminate successfully only if P, W_1, W_2 and W_3 all commit their objects.

The commit algorithm presented here is known as the two–phase commit algorithm [Gray 78]. There are several ways of optimising this algorithm, some of which are discussed in [Mohan and Lindsay 83].

5.4 Another Look At Crash Resistance

Robust actions described in the section on non–distributed programs have the property that a crash of the node causes all the actions in progress to be aborted (except those that are commiting). Their distributed counterparts have a similar property in that a crash of any node belonging to the group of processes executing an action causes the abortion of that action. We will discuss how such type *(1)* programs can be utilised for constructing type *(2)* programs, so that a computation can complete normally despite crashes. This is an attractive proposition for a distributed program if a distributed system contains redundancy in the form of services provided.

Assume that we wish to construct a program to run on node N_1 that calls operations of some recoverable object, A, on node N_2, and that there is another object, B, on node N_3 which provides similar operations to A. We require the property that if N_2 crashes in the middle of the computation in question (or becomes unavailable from N_1 due to some communications breakdown), then rather than aborting the whole computation, only the side effects produced on A should be undone and the computation continue by accessing object B instead of A. We can make use of nested concurrent actions to achieve this property, as indicated below.

(1) The process (say P) running the main program as a robust action on N_1 forks a process, Q, and waits.

(2) Process Q starts a new action and establishes a recovery point; it creates a server process, W_1, on N_2 to receive remote calls for A.

(3) If a call to A fails then Q performs local recovery and aborts; P now continues.

(4) There are two possibilities at node N_2: either N_2 has crashed, in which case there is no W_2, or W_2 is running. If the latter is true, then W_2 will eventually abort (see the algorithm for a worker given in the previous subsection).

(5) Process P can fork another process to perform operations on object B.

We will gloss over the details of recovery data management for concurrent actions as they are not essential to understand the point that concurrent actions can be aborted independent of the enclosing action. Naturally, the above idea can be applied to any degree of nesting of actions.

Returning to our example, if the home node of the robust action (N_1) crashes, then the entire action will be aborted. If this is not deemed desirable, then a distributed checkpointing facility will be required so that the intermediate state of the entire computation can be saved on stable storage. We will not pursue this topic further here.

6. Some Distributed Object–Based Systems

In this section we will briefly describe some prototype distributed systems that have been either developed recently or are in the process of development. All these systems make use of robust objects and atomic actions to provide an environment for reliable distributed computing. As yet, little operational experience is available to be able to make comparative evaluation of the advantages and disadvantages of the various specific techniques employed in these systems.

6.1 Argus

Argus [Liskov and Scheifler 83] is a programming language and system designed at MIT that supports the building of robust distributed programs. In Argus, programs are implemented out of one or more modules known as *guardians*. Each guardian consists of data objects and processes for manipulating those objects. An object is only allowed to belong to one guardian. Objects within a guardian can be directly manipulated by processes internal to the guardian, but not by other guardians. Instead, access to these objects is provided by a set of *handlers* (procedures) that can be called by other guardians. Handler calls are remote procedure calls with each call executing as a nested atomic action. Furthermore, all parameter passing is by value making it impossible to pass an object reference in a handler call. A guardian resides at a single site only and is crash–resistant. If a guardian's site crashes then all processes are lost but a subset of the guardian's objects (its *stable state*) survives and will be restored when the site recovers.

Argus provides a set of basic atomic data types such as arrays and records, as well as the ability to construct user–defined atomic data types. An action starts at one guardian but can spread to others by making handler calls. Language features have been provided for constructing nested and concurrent actions.

6.2 ISIS

The ISIS [Birman 85] project from Cornell University aims to produce fault–tolerant implementations of objects automatically from fault–intolerant program specifications. The resulting objects are then known as *resilient* (or more precisely, *k–resilient*) objects. ISIS works by replicating both the code and data of an object at least k+1 times, while ensuring that the replicated program behaves exactly like a non–replicated program of the original specification. Resilient objects are represented at a set of k+1 sites by *components* that are capable of executing requests sent to them via remote procedure calls. Each request is handled as a separate nested atomic action. The use of replication limits the requirements for stable storage since a failed component can recover its state from any of the other operational components. One component is known as the *coordinator*, while the remaining components are known as *cohorts*. It is the coordinator who is responsible for executing the request and returning the result to both the caller and all the cohorts. If the coordinator fails one of the cohorts becomes the new coordinator and processing of the request continues. The individual components are statically ranked such that the choice of which cohort becomes the new coordinator is easy to make.

Objects in ISIS exhibit the following properties:

(1) *Consistency.* The external behaviour of a resilient object is identical to that exhibited by an equivalent non–distributed object which executes requests serially and to completion with no interleaving.

(2) *Availability.* Providing that no more than k sites holding components of an object fail simultaneously the other operational components of the object will continue to process requests.

(3) *Progress*. Providing no more than k sites of a resilient object fail, operations on it are executed to completion.

(4) *Recovery.* Failed components restart automatically.

ISIS is built on top of a communication layer that implements a set of broadcast facilities, which provide a variety of guarantees about the order in which messages will be delivered. A failure detection mechanism is also integrated into this layer. ISIS objects are coded in an extended version of the C programming language. The extensions allow internal concurrency, together with atomic action control, remote procedure calls, and other facilities.

6.3 Profemo

The Profemo [Nett et al. 85] object–based distributed system aims to use specially designed hardware to provide support for object access, storage, and atomic action management. The object space, the regions where objects or information about objects exist, is divided into four distinct parts. The *active* and *recovery* spaces exist in virtual memory, the former containing objects in use by the system, the latter contains recovery information required to recover from action failures. The *permanent* and *log* spaces are implemented on stable storage to provide tolerance against both site and media faults. Objects held in the permanent space are effectively filed, and may be used by both local and remote programs transparently. The log space contains information required to survive a site failure, and may contain newly committed objects which will eventually be propagated to the permanent space. The recovery and object management are all supported by the operating system kernel to improve performance.

Support for nested atomic actions is also built into the kernel and is aided by dedicated hardware. Two phase locking is used, but the shrinking phase is not made instantaneous; objects no longer required by an action may be released before the action terminates. This has the advantage of increasing possible concurrency, but the disadvantage of cascade roll back as mentioned in Section 2. In order to be able to determine the extent of cascade roll back required, it is necessary to record dependencies between actions. Such dependencies are maintained in a *recovery graph*. Profemo provides hardware support for recovery graph maintenance and for other aspects of recovery management.

6.4 Arjuna

The goal of the Arjuna project is the development of a distributed object–based system for programming with atomic actions. The project exploits much of the work done at Newcastle on reliability and distributed systems [Shrivastava 85]. In particular, it employs a previous RPC design [Panzieri and Shrivastava] which has been modified to incorporate multicasting facilities, and is using multicasting RPCs to implement commit protocols and manage replicated data. Rather than provide a new language or operating system with support for robust objects and actions, Arjuna is exploiting the type inheritance facility provided by the implementation language, C++ [Stroustrup 86], to add both recoverability [Dixon and Shrivastava 87] and concurrency control [Parrington and Shrivastava 87] to objects. Nested atomic actions which use these objects may be employed along with stub–generated RPC invocations to provide reliable distributed computations.

Recoverable objects can be constructed out of both recoverable and unrecoverable objects. Rather than provide an explicit recovery subsystem, the objects themselves are responsible for

initiating the process of recording recovery information, and operate in conjunction with the program which is using them. The atomic action model is also being extended to allow the possibility of long running actions (LRA) which release objects before the action is committed [Shrivastava 82]. The system is being developed on top of the UNIX[2] operating system.

7. Concluding Remarks

The approach adopted by this paper was first to develop a framework for constructing robust non–distributed programs and then to extend that framework to encompass distributed programs. We have presented what may be termed an object based recovery approach, whereby objects are responsible for implementing backward recovery and crash resistance. A number of object–based systems making use of the concepts reviewed here were discussed to indicate to the reader the wide scope of activity in the field of reliable distributed computing.

Acknowledgments

This paper has benefited from critical comments by: Tom Anderson, Brian Randell, and Josephine Anyanwu. The work reported here is supported in part be a SERC/Alvey grant in Software Engineering.

References

Anderson et al 78

Anderson, T., Lee, P.A. and Shrivastava, S.K. "A Model of Recoverability in Multilevel Systems," *IEEE Transactions on Software Engineering*, Vol. SE–4, No. 6, pp. 486–496, November 1978.

Bernstein and Goodman 81

Bernstein, P.A. and Goodman, N. "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 182, pp. 185–221, June 1981.

Best and Randell 81

Best, E. and Randell, B. "A Formal Model of Atomicity in Asynchronous Systems," *Acta Informatica*, Vol. 16, pp. 93–124, 1981.

Birman 85

Birman, K. P. "Replication and Fault Tolerance in the ISIS System," *Proceedings of 10th Symposium on Operating Systems Principles*, ACM Operating Systems Review, Vol. 19, No. 4, pp. 79–86, December 1985.

Birrell and Nelson 84

Birrell, A.D. and Nelson, B.J. "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39–59, February 1984.

Cristian 82

Cristian, F. "Exception Handling and Software Fault Tolerance," *IEEE Transactions. on Computers*, Vol. C–31, No. 6, pp. 531–540, June 1982.

Dixon and Shrivastava 87

Dixon, G.N., and Shrivastava, S.K. "Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems," *Proceedings of the 6th Symposium on Reliability in Distributed Software and DataBase Systems*, Williamsburg, pp. 107–114, March 1987.

2. UNIX is a registered trademark of AT&T in the USA and other countries

Eswaran et al 76

Eswaran, K., Gray, J.N., Lorie, R. and Traiger, I. ”The Notions of Consistency and Predicate Locks in a Database System,” *Communications of the ACM*, Vol. 19, No. 11, pp. 624–633, November 1976.

Gray 78

Gray, J.N. ”Notes on Database Operating Systems,” *Lecture Notes in Computer Science*, Vol. 60**,** Springer–Verlag, pp. 398–481, 1978.

Hamilton 84

Hamilton, K.G. ”A Remote Procedure Call System,” Technical Report No. 70, Computing Laboratory, University of Cambridge, 1984.

Horning et al. 74

Horning, J.J., Lauer, H.C., Melliar–Smith, P.M., and Randell, B. ”A Program Structure for Error Detection and Recovery,” *Lecture Notes in Computer. Science.*, Vol. 16, Springer–Verlag, pp. 177–193, 1974.

Lamport et al. 82

Lamport, L., Shostak, R. and Pease, M. ”The Byzantine Generals Problem,” *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 382–401, July 1982.

Lampson and Sturgis 81

Lampson, B. and Sturgis, H. ”Atomic Transactions,” *Lecture Notes in Computer Science*, Vol. 105, Springer–Verlag, pp. 246–265, 1981.

Lee and Anderson 85

Lee, P.A. and Anderson, T. ”Design Fault–Tolerance,” in *Resilient Computing Systems*, ed. T. Anderson, Collins, pp. 64–77, 1985.

Liskov and Scheifler 83

Liskov, B. and Scheifler, R. ”Guardians and Actions: Linguistic Support for Robust, Distributed Programs,” *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, pp. 381–404, July 1983.

Mohan and Lindsay 83

Mohan, C. and Lindsay, B.G. ”Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions,” *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, Montreal, pp. 76–88, August 1983.

Moss 82

Moss, J.E.B. ”Nested Transactions and Reliable Distributed Computing,” *Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems*, pp. 33–39, Pittsburgh,July 1982.

Nett et al. 85

Nett, E., Großpietsch, K., Jungblut, A., Kaiser, J., Kröger, R., Lux, W., Speicher, M., and Winnebeck, H. ”Profemo: Design and Implementation of a Fault Tolerant Distributed System Architecture,” *GMD–Studien*, Nr. 100, 1985.

Panzieri and Shrivastava

Panzieri, F., and Shrivastava, S.K., ”Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing,” to appear in *IEEE Transactions on Software Engineering*.

Parrington and Shrivastava 87

Parrington, G.D., and Shrivastava, S.K. ”Implementing Concurrency Control for Robust Object Based Systems,” (to be published).

Shrivastava and Banâtre 78

Shrivastava, S.K. and Banâtre, J.–P. ”Reliable Resource Allocation Between Unreliable Processes,” *IEEE Transactions on Software Engineering,* Vol. SE–4, No. 3, pp. 230–241, May 1978.

Shrivastava 81

Shrivastava, S.K. ”Structuring Distributed Systems for Recoverability and Crash Resistance,” *IEEE Transactions on Software Engineering,* Vol. SE–7**,** No. 4, pp. 436–447, 1981.

Shrivastava 82

Shrivastava, S.K. ”A Dependency, Commitment and Recovery Model for Atomic Actions,” *Proceedings of 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems,* Pitsburgh, pp. 112–119, 1982.

Shrivastava 85

Shrivastava, S.K., ed. *Reliable Computer Systems*, Texts and Monographs in Computer Science, Springer–Verlag, 1985.

Stroustrup 86

Stroustrup, B. *The C++ Programming Language,* Addison Wesley, 1986.

Svobodova 84

Svobodova, L. ”Resilient Distributed Computing,” *IEEE Transactions on Software Engineering,* Vol. SE–10, No. 3, pp. 257–268, May 1984.