Abstract:

An object oriented model of computation is used to discuss recoverability and crash resistance issues in distributed systems. The execution of a program at a node of the system can give rise to a hierarchy of processes executing various parts of the program at different nodes. Recoverability and crash resistance properties are needed to ensure that such a group of processes leave the system state consistent despite faults in the system.

Structuring Distributed Systems for Recoverability and Crash Resistance

By

S.K. Shrivastava
SHRIVASTAVA, Santosh Kumar.


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1979.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 149.)

Added entries
UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)

Library of congress:
Dewey (17th) 001.6425
U. D. C. 519.687

Suggested keywords
DISTRIBUTED SYSTEMS
CONSISTENCY
COMMITMENT
RECOVERABILITY
MESSAGE PASSING
CRASH RESISTANCE
ATOMIC ACTIONS
CONCURRENCY
BACKWARD ERROR RECOVERY
EXCEPTION HANDLING
SECURE STORAGE

Abstract

An object oriented model of computation is used to discuss recoverability and crash resistance issues in distributed systems. The execution of a program at a node of the system can give rise to a hierarchy of processes executing various parts of the program at different nodes. Recoverability and crash resistance properties are needed to ensure that such a group of processes leave the system state consistent despite faults in the system.

About the author

Dr. Shrivastava joined the Computing Laboratory of the University of Newcastle upon Tyne in August 1975, where he is currently a Lecturer. Prior to that he was with the Plessey Co. Ltd.
INTRODUCTION

Consider a computing system consisting of a number of autonomous computers (referred to as nodes) connected by a communication system that allows the various nodes to exchange information with each other. Each node of such a system will provide one or more services (e.g. data retention, document printing, compiling, text editing) and a user computation running on any node can make use of such services by suitable use of the communication system. Such a system will be called a distributed system. When a user computation running on a given node invokes a legitimate service call to some other node, there can be many reasons why that service might not be available; for example, the communication link between the nodes is perhaps faulty or the server node is temporarily out of order and so on. For these, and many other reasons, it is quite possible for a user computation to arrive at such a state from where further meaningful progress is not possible. Such a state affairs can be highly undesirable in a system where node services are constantly being shared between various computations — since a computation that has not terminated satisfactorily could leave part of the system state inconsistent. Under the assumption that, if a computation (that begins when the system state is consistent) terminates properly then it will leave the (possibly new) system state consistent, we require the following abstract property from all programs to guarantee consistency in the presence of faults: the computation invoked for a program either terminates or that invocation has no effect on the system state. The above abstraction can be maintained if the system can be structured to provide recoverability such that the current state of an unsatisfactory computation can be abandoned in favour of a prior state.

In this paper we shall examine some of the fundamental issues that are involved when this so-called backward error recovery facility //1// is to be provided in a distributed system. The contributions of this paper are not that it presents novel techniques for implementing the above form of recoverability - excellent papers exist on this topic (e.g. //2,3//) - but that it attempts to describe the underlying concepts on the means of achieving recoverability in distributed systems. These ideas, it is hoped, will not only help the reader in critically reviewing any planned or existing systems, but will also provide a framework for formulating arbitrarily complex recovery strategies and their implementations.

The rest of the paper is divided into four sections. In the first section we introduce an object oriented model of computation that will be assumed throughout. In the second section we discuss recoverability issues that arise while in the third section we discuss how a distributed system can be made crash resistant. Finally, in the fourth section directions for future work are discussed.
A MODEL OF COMPUTATION

We shall view the system as consisting of a collection of abstract objects. An abstract object is characterised by an abstract data and a set of abstract operations that manipulate the data - these operations are the only means of data manipulation. The term object manager will be used to refer to the implementator of the corresponding abstract object. An object manager maintains a concrete representation of the abstract data; an abstract operation then corresponds to the manipulation of the concrete data by programs provided by the manager. Needless to say that an object manager itself can use other abstract objects as a part of its concrete representation - thus complex objects can be built hierarchically. All this, we hope, is intuitively obvious to readers generally familiar with these by now well accepted software engineering concepts. It should also be obvious when we say that if an abstract object is recoverable then it is indeed because of the object manager which provides this abstraction. What is not so obvious is how an object manager may provide this abstraction in a distributed system; this is one of the subjects of the paper.

We classify objects into two categories: active and passive. An active object is in fact a computation - the activation of a program; we shall henceforth use the commonly accepted term process to refer to an active object. Processes are the users of passive objects.

As is well known, in any system a hierarchy of abstract interfaces (or levels) can be discerned. Any given level is characterised by the abstract objects made available on that level together with their associated operations for their manipulations. We say that an object is in a consistent state (or simply consistent) if it satisfies certain invariant properties. An interface is said to be in a consistent state if all of its objects are consistent. Consider now a number of programs P1,P2,...,Pn written to run on a given interface (level) Li. These programs are in fact providing abstract operations for the next interface (Li+1) - the higher level of abstraction. In order that the invocations of these programs preserve the consistency of interface Li+1, these programs must satisfy the following two properties:

i) when execution of any program Pi is considered in isolation, then Li+1 remains consistent.

ii) when programs Pi,...,Pm are being executed concurrently, then these executions are interference free.

Programs that satisfy property (ii) are said to implement atomic actions /4/. These two properties are necessary and sufficient for the maintenance of consistency in a concurrent environment. We shall not prove this here; the interested reader is referred to /3,4,5/. From now on we shall only consider programs that implement atomic actions.
If programs are manipulating objects that are shared, then it is necessary for them to follow some appropriate locking protocol on these objects //3,5/>. We shall assume that some such protocol, also necessary to obtain the kind of recovery to be discussed in the next section, is being observed.

We shall now consider a distributed system at a sufficiently high level of abstraction such that processes at different nodes have message sending facilities for interprocess communications with the following properties. A process can send a message to a named receiver at a given node using a 'send' primitive. Successful execution of this primitive implies that the message has been delivered to the receiver. Either a 'time out' or 'process missing' exception can be raised during the invocation of a 'send'. The former exception implies inability to send the message (possibly due to communications failure) and the latter exception implies that the receiver process is no longer there. A process can also wait for a message; a time out exception is associated with this 'wait'. We assume that the following technique is used by every process that wants a service from some other process //2//:

\[
\text{send message(processname, node, message)} \\
\hspace{1cm} \text{[for all exceptions: take recovery actions]} \\
\text{wait: receive message(processname, node, result)} \\
\hspace{1cm} \text{[for all exceptions: take recovery actions]} \\
\hspace{2cm} \text{analyse result; if unsolicited message} \\
\hspace{2.5cm} \text{then goto wait else "expected response"} \\
\hspace{3cm} \text{take actions based on result}
\]

We shall use the following concise notation for the above steps:

\[
\begin{align*}
\text{send message(processname, node, message)} & & \text{recovery steps} \\
\text{receive message(processname, node, result)} & & \\
\end{align*}
\]

- "take actions based on result" -

The interesting issue that arises is how can a process executing a program at one node access objects at other nodes? We shall assume the following technique for achieving the above aim:

\[
\begin{align*}
\text{create worker(worker1, N1)} \\
\text{send message(worker1, .....)} & & \text{recovery steps} \\
\text{receive message( ..... )} & & \\
\end{align*}
\]
The above program indicates how a process (call it master) executing that program will access objects at a remote node \( N_i \). We assume that a primitive operation for creating a process at a given node is available; so a process (named worker\(_i\)) is first created at \( N_i \). Next the master sends specific requests to worker\(_i\) for performing operations on objects at \( N_i \) on its behalf. When a worker process invokes an abstract operation (on behalf of the master) then further worker processes might be created on its behalf since the program implementing that operation might need access to objects at other nodes (clearly this situation implies that the abstract object the worker is accessing has its concrete representation on objects, some or all of which are at remote nodes). Thus the execution of an atomic operation can give rise to a hierarchy of processes with master-worker relationships as shown in Figure 1.

![Figure 1: Process hierarchy](image)

This concludes our discussion on the model of computation. In the next section we consider some of the issues that arise when recoverability is required.

**RECOVERABILITY ISSUES**

Recoverability is desired in order that consistency be maintained in the presence of faults. As stated in the introduction, by recoverability we shall mean backward error recoverability - the ability to restore an earlier state of a computation (for a discussion on the importance of backward error recovery and its relationship to the so-called forward error recovery - exception handling- technique, the reader is referred to /1/). Using the terminology and concepts developed for an earlier paper /6/, for a process to have the abstraction of recoverability at a given level of abstraction requires that:

1. the process can establish a recovery point - thus indicating the start of a new recovery region (see Figure 2). This implies that in case an error is detected, the states of any recoverable objects modified in that region can be restored to those at the beginning of the region. The following notation will be assumed:
establish recovery point (control)

where 'control' specifies the point where the flow of control should be after state restoration (e.g. for a recovery block //7//, 'control' could be the starting address of the next alternative).

(ii) the process can discard a recovery point, thus indicating the end of a recovery region. As shown in Figure 2, a process can successively establish recovery points giving rise to nested recovery regions //6//.

```
   "process"
     recovery region1  
                     
               establish recovery point1();
                     
               recovery region2  
                     
               establish recovery point2();
                     Time
                     
               discard recovery point2;
                     
               discard recovery point1;
```

Figure 2: Multiple Recovery Points

The establishment of a recovery point entails notification to all those object managers (whose services will be used by the process) that are providing the abstraction of recoverability (i.e. are providing recoverable objects) so that the managers can commence recording appropriate recovery data. When an error is detected, all these managers must be appropriately notified so that they can use their recovery data for providing the abstraction of recovery. The following three notifications must be handled by an object manager: (i) record recovery data, (ii) discard recovery data, and (iii) recover. Note that it is being assumed that after a 'record' notification, any number of operations on that object can be invoked - the effects of all of these can be undone by the corresponding 'recover' notification.

For recovery to be automatic, it is clear that these notifications must be performed automatically on behalf of the process in question. A logical organisation for how this may be done will now be briefly discussed (for the time being only access to local objects will be considered).
erp establish recovery point
drp discard recovery point

"Process Q"

Manager of object M

begin
  erp();
  ...
Li "Program Pj"

  ...
  M.OP();
  ...
end

Li-1  Li  L0

"Program Pi"

  ...
  drp;
  ...
end

Interpreter IO

Figure 3: A hierarchy of interfaces.

Referring to Figure 3, it is assumed that interface L0 (that is being maintained by a set of programs IO known as the interpreter for L0) provides some recovery facilities (see below) to programs that run over it. A number of object managers have been programmed on L0 thereby successively extending L0 to Li "66". Thus a program Pj can not only use most of the operations of L0, a number of additional operations - those provided by the programmed managers - have also been made available to it. Consider now the execution of some program - say Pj. Every invocation of an operation on interface Li is first examined by IO which determines whether that operation is directly supported by itself or if not, which of the managers (also referred to as extensions in "66") does support it - the operation of that particular manager is then invoked by IO ('M.op(...)' in this particular case). We shall now make the assumptions that L0 supports the abstraction of processes and that IO maintains some data per level for the purposes of recovery for each process (we shall refer to this data by processname.level.recovery data). Then, when a process such as Q executes an operation 'establish recovery point', this results in IO making an appropriate entry in Q.level.recovery data - thus signifying the start of a new recovery region at that level. Note that recovery regions of programs such as Pj and Pi are independent from each other, despite the fact that they 'belong' to the same process (Q in Figure 3); this is because they are at different levels of abstractions. The invocation of an operation such as 'M.op(...)' results in IO executing the following program (for simplicity, in the following algorithms it has been assumed that IO itself has no recovery capability and also that it is not providing any recoverable objects):
begin
if M is recoverable then
search the current recovery region of Q.level.recovery data for object name M;
if not found then
begin record the name M in Q.level.recovery data;
invoke M.record recovery data (j,level);
end
invoke M.op(...)
end

In the above program, 'i' will be the level number of Q when it executes M.op(...). Whenever IO invokes recovery data associated operations (on behalf of the caller) of an object manager, the level number of the caller is also implicitly supplied. This number may be utilised for the management of recovery data by a manager in a manner similar to that done by IO. The first parameter of 'recovery recovery data' operation specifies the number of the current recovery region.

What happens when an error is detected during the execution of an operation of an object manager? We assume the following methodology for the treatment of erroneous conditions: an object manager can provide a number of routines - exception handlers - for coping with any anticipated abnormal situations. If an error is detected during the execution of a program of an object manager for which no handler is available, then the backward recovery capability of that program (if any) will be invoked and after which any fault treatment measures associated with that program will be undertaken. For example, if the program is making use of recovery blocks //7//, then an alternative program will be executed. If no fault treatment measure exists or equivalently, the available measures have proved inadequate (e.g. all the alternatives have failed the acceptance test) then a failure notification signifying a failure of that program is signalled to the caller so that the caller can invoke its own recovery measures, if any //8,9/>. Failures of critical programs such as those concerned with recovery data management can be quite serious and may eventually require that crash recovery procedures be invoked (to be discussed subsequently). We assume that two primitive operations are provided by IO to facilitate the above discussed error handling: an 'abort(i)' primitive whose execution results in the invocation of backward error recovery to the ith recovery point and a 'fail' primitive the issuing of which results in IO executing the following program:

begin
signal failure to the caller of the failed program
end

Needless to say that the above error handling ideas discussed with reference to object managers are equally applicable to the programs of IO. In the rest of this paper, unless otherwise stated, whenever we say 'an error is detected' we shall mean 'an error is detected for which no exception handler is available'.
The invocation of operation 'discard recovery point' by Q results in IO executing the following program:

```
begin
  for all object names recorded in the current region of Q.level.recovery data do invoke object name.discard recovery data (level)
  ------ delete recovery data no longer needed ------
end
```

Finally, when Q executes an abort(j) primitive, this results in IO executing the following program:

```
begin
  prepare the set of object names that appear in Q.level.recovery data in region j to the current region;
  for all the elements of the set do invoke objectname.recover(j,level); delete recovery regions j to the current one
end
```

The above discussion has so far assumed the existence of only one process. We shall now consider how concurrency affects the management of recovery. To start with, it is clear that a program implementing an atomic action should be made recoverable such that when an error is detected, capability for undoing the entire actions of that program exists. Bearing this in mind, the following structure suggests itself:

```
an atomic action with recovery = "begin ----> establish recovery point1()
  action"
  establish recovery point2()
  establish recovery point1()
  discard recovery point1
  "end
  action" ----> discard recovery point1
```

A recovery point is established at the start of an atomic action. Because of our earlier assumption that an action that terminates properly maintains consistency, the end of an action represents the moment when any recovery data associated with that action can be discarded. As shown above, any number of intermediate recovery points can be established and discarded in the middle of a program.

When the outermost recovery point is discarded at the end of an action, commitment occurs in that the state at the start cannot be re-created. All the objects modified by an action are then said to be in a committed state. An important issue arises regarding concurrent
execution of atomic actions: should an action be allowed to access objects that are in uncommitted states because other actions have yet to commit? If the answer is yes, then, in general, it is possible that recovery of an action may require that some other uncompleted actions be recovered which in turn may require further actions in progress be recovered. This 'domino' effect //10// can be avoided and recovery of actions made independent from each other if the answer to the question is no - thus requiring that an action only access objects that are initially in committed states. From the point of view of implementation, this requires that unlocking of shared objects be performed on or after the end of that action //3,5//. This results in a lesser degree of concurrency than the first case - this is the price paid for simplicity in recovery requirements. For the purposes of this paper we shall assume this simple recovery strategy and that every object manager and the interpreter - IO in Figure 3 - maintain recovery data to provide this unilateral recovery of an action. We further assume that every object manager provides appropriate looking operations (which are recoverable if the manager provides recoverability) and that all programs implementing atomic actions appropriately lock objects. Locking of objects so as to guarantee atomicity is a research topic in its own right - as are the topics of deadlock detection and prevention. Fortunately what we have to say here is orthogonal to any locking protocol used, we shall therefore assume any appropriate locking protocol:

establish recovery point();

\[
\text{atomic action with recovery} \quad \smiley \\
\quad \text{appropriately lock all shared objects to be accessed}
\]

\[
\text{commit operations} \quad \left\{ \\
\quad \text{discard recovery point;} \\
\quad \text{unlock all the locked objects; for all the workers created do} \\
\quad \text{delete worker(workernname,node)};
\right.
\]

Note that no matter which protocol is used, locks must be released only during commitment. The last commit action consists of destroying all the worker processes created in that action. A 'delete' primitive is assumed to exist for this purpose. If an action has no recovery capability then establish and discard operations will be absent.

Assuming the above framework for managing concurrency, two issues now remain to be discussed in this section on recoverability, namely (i) steps taken by an object manager to provide the abstraction of recoverability, and (ii) the treatment of remote objects. We shall consider these two issues in turn.

The object M of Figure 3 will be used for illustrative purposes. Let us assume that the concrete implementation of M is on three local objects A, B and C and that A and B themselves are recoverable. The task of the object manager is thus to provide a recoverable object M constructed out of some recoverable and unrecoverable objects. The following figure shows the various
programs and data structures of the manager of M, where a circle represents abstract data.

It has been shown that there can be two ways of supporting

\[
\begin{array}{c}
\text{record} \quad \text{recovery} \quad \text{discard} \\
\text{data} \quad \text{data} \quad \text{data} \\
\text{Pi} \quad \text{A} \quad \text{B} \quad \text{C} \\
\text{recover()} \\
\text{other programs implementing} \\
\text{other operations of M} \\
\end{array}
\]

\begin{align*}
\text{Pi} & \quad \text{A} \quad \text{B} \quad \text{C} \\
\text{Li} & \quad \text{Pj} \\
\text{LO} & \end{align*}

**Figure 4. Data structures and programs of an object manager.**

recoverability of objects: by either using the disjoint recovery scheme or the inclusive recovery scheme //6//. The interpreter IO is said to support the disjoint recovery scheme if, in the above figure, objects A and B appear as recoverable 'locally' - that is only to the programs of the manager of M such as Pi. Thus, when an error is detected in Pj, IO will automatically execute the recover program of the manager of M, the same is not done for those of A and B. It is entirely the responsibility of the manager of an object to provide the abstraction of recoverability. In this particular case, programs 'record' and 'recover' could be (assuming for simplicity that recovery data is collected straight away rather than incrementally):

\[
\begin{align*}
\text{record recovery data } \equiv & \\
& \text{invoke 'read' operations on} \\
& \text{A, B and C and store data in} \\
& \text{a new region of 'recovery data'}
\end{align*}
\]

\[
\begin{align*}
\text{recover(j) } \equiv & \\
& \text{invoke 'write' operations on} \\
& \text{A, B and C, using data stored in} \\
& \text{the jth region of 'recovery data'}
\end{align*}
\]

It can be seen that we have been implicitly assuming the disjoint recovery scheme so far (see the previous discussion on IO).

The interpreter IO is said to support the **inclusive recovery scheme** if, in the above figure, objects A and B appear recoverable
'globally'. Thus, when an error is detected in $P_j$, I0 will execute 'recover' operations of the managers of $M$, $A$ and $B$. This means that an object manager, in order to provide recoverability, need only be concerned with the unrecoverable objects it is managing:

$$\text{record recovery data} \equiv$$

invoke 'read' operations on $C$ and store data in a new region of 'recovery data'

$$\text{recover}(j) \equiv$$

invoke 'write' operations on $C$ using data stored in the $j$th region of 'recovery data'

The above example would seem to indicate that the inclusive scheme provides an easier means of structuring recoverable objects than the disjoint scheme. However, as has been discussed elsewhere //6//, inclusive scheme does need the features of disjoint scheme (i.e. localised recovery) during the execution of programs with recovery features that manipulate recovery data (this is because if recovery data were stored on recoverable objects, then when an error is detected in $P_j$, this data must not be restored), thus its implementation by I0 is much more complex. In a distributed system, an implementation of inclusive scheme is even more complex because concrete objects (such as $A$, $B$ and $C$) can be on different nodes; thus an interpreter at one node can not easily invoke 'record' and 'recover' operations of objects at other nodes. For these reasons, the disjoint recovery scheme appears as most suitable in a distributed environment. In the rest of the paper, unless otherwise stated, we shall be assuming such a recovery scheme. To exploit the recoverability of existing objects however, is an important advantage of the inclusive scheme; we shall later present a method whereby a master process can exploit the recoverability of remote objects in a manner that models the inclusive scheme.

It has been assumed here that object managers perform 'update in place' so that 'record recovery data' operations are for recording prior states. An opposite scheme is also possible whereby 'record recovery data' operations in fact record changes to the current state and the 'discard recovery data' includes operations to make these changes to the prior state. Which scheme to use is entirely up to the designer of an object manager and is of no concern to the users of that object; the former scheme is being assumed in this paper purely for the sake of illustration.

We now turn our attention to the treatment of remote objects. We shall consider two cases to explain the structuring to be proposed: (i) program $P_j$ accesses an object $Z$ located at a remote node (say $N_1$); and (ii) object $M$'s concrete representation is also
The remote handler object records on behalf of each process, the names of worker processes and their nodes that are created during the execution of a recoverable atomic action at a given level. Thus for action Pj, the handler will record, in a region specially maintained for process Q, wi,Ni. When Q starts executing Pj, the handler will record (in a different region) wl,Nl and wk,Nk. If the execution of Pj ends, wl and wk will be destroyed and when the execution of Pj ends, wi will also be destroyed. Note how messages for establishing recovery points are automatically sent to workers. Thus, if after creating a worker, a master progressively establishes n recovery points, n messages to that effect will be sent to the workers in appropriate order by the remote handler. For simplicity, it has been assumed that the above programs have no recovery capability except to signal failure when abnormal conditions occur. Occasionally, this might leave worker's list structure of the current recovery region of a process in an inconsistent state; an abort operation by the caller should normally be sufficient to restore consistency. However, if operations such as recover fails, only crash recovery procedures (see later) can restore consistency.

Next we shall study the actions of a worker. A worker acts as a command interpreter for its master. Logically, therefore, a worker can provide the abstraction of recoverability to its master exactly as the interpreter to and other object managers at the master's node do - namely by appropriately recording recovery data and to use this data when recovery is desired. The point to note is that backward recovery of a master can be supported by a worker by its normal forward actions.

An undesirable property of this is that a worker has to implement the recoverability of all of the objects it accesses on its master's behalf - even if these objects are recoverable (this is of course, the disadvantage of the disjoint recovery scheme). A further disadvantage - and this is more serious - is that a worker, in general, does not 'know' the objects it will be called upon to access; implementing recoverability under such a condition can be very difficult. The question then arises as to how can a worker effectively use the recoverability features of its local node? One way to achieve this goal is for a worker to support the recoverability of the master by invoking its - the worker's - own recoverability:

```
"actions of a worker process"
cycle
  receive message( ); [time out exception: kill]
  if message is establish recovery point then
    {result:=OK; establish recovery point (resume)}
  else {execute command in the message; prepare answer?}
    [for all exceptions: result:=exception type; goto resume]
  resume: send message(master,Nmaster,result)
    [for all exceptions: kill]
end "cycle"
```
When a worker receives an establish recovery point command, it responds by creating its own recovery point, with flow of control at 'resume'. If the received command is 'abort', the worker will execute this command - thus invoking its own recovery. As a result of this, all the recoverable objects updated by the worker are restored appropriately and the worker sends a message to master indicating that the recovery has been performed. This scheme makes the program of a worker simple and the existing recovery facilities are efficiently utilised as in the inclusive recovery scheme. Note that it is now a master's responsibility to maintain, if so desired, any recovery information for those objects that are unrecoverable at a worker's node.

The worker reports all the abnormal conditions encountered to its master. However if during sending or receiving of a message exceptions are raised, then this is taken to mean either a collapse of the worker-master communication facility or the master node. The worker's response is to destroy itself using the kill primitive provided by the underlying interpreter TO which can implement it as follows:

```
begin
  construct the set of object names (if any) recorded in calling process.level.recovery data in recovery region 1 to the current one;
  for all these object names do
    invoke objectname.recover(1,level);
    delete all the information maintained for the calling process
end
```

We now consider the second case mentioned earlier, namely, how can an object manager map the abstract data it is providing on to its objects, some or all of which are on remote nodes. The discussion to follow will use the example of object M at node Nlocal whose concrete representation is on recoverable object A at NK, recoverable object B at Nlocal and unrecoverable object C at Nl. The scheme we have presented for coping with remote objects can be used effectively by the manager of M as follows: (again we are using the simple scheme of recording all recovery data at once rather than incrementally as our major concerns are with logical issues rather than those concerned with efficiency):

```
record recovery data 

create workers at NK and Nl;
use 'read' operations on A,B and C and record data in recovery data;
destroy workers at NK and Nl;
```

other operations similarly create and destroy workers (it is certainly possible to optimise the number of times workers are created and destroyed).
An alternative technique that models the inclusive scheme is also possible whereby worker processes can be created to exploit the recoverability facilities of their nodes. Under this scheme, worker processes are created by an object manager at the beginning when its 'record' operation is called for the first time by a process. All the operations of the manager use these workers which are destroyed in the 'discard' operation when the outermost recovery region is discarded. Algorithms needed under this scheme should be fairly obvious to the reader; a sample algorithm is given below as an example:

\[
\text{record recovery data} \equiv \begin{array}{c}
\text{if workers not created for calling process then create workers at Nk and NL;}
\text{send message to establish recovery point to worker at Nk;}
\text{invoke read operations on B and C (using worker at NL) and record data in recovery data;}
\end{array}
\]

Note that if the local node above supports the inclusive recovery scheme then there would be no need to record any recovery data for B. Also that it is much more difficult in this technique – and this follows as the inclusive scheme is being modelled – for programs of an object manager to use recovery features for local recovery (this can be appreciated if the reader considers the problems encountered, if in the above program, recovery points are desired for recovery of the program itself). Such problems are not encountered in the previous case: the object manager can record recovery data on recoverable objects and use recovery features in its programs without any complications. Lastly, it should be noted that the framework proposed here is sufficiently general in that objects that store recovery data need not be local.

**CRASH RESISTANCE ISSUES**

A distributed system is subject to independent failure modes of its components – communications equipment and computers. The communication facility between a pair of nodes is said to have 'crashed' if the nodes, while perhaps able to exchange messages with other nodes, can not do so with each other. A node crash occurs if the interpreter of that node (IO) can not maintain the interfaces (L0, ..., Li) in 'acceptable' states – where the acceptability criterion is chosen by some external monitoring agency – (say an operator). We shall make the following assumption when a node crashes: all the data stored in the system (except that stored on a secure storage, see below) is in an inconsistent state. Consider now our previous example of a process (Q) executing a program Pj that accesses a local object M at Nlocal and a remote object Z at Ni. Suppose during the middle of the execution of Pj, node Ni crashes. While it is certainly possible to restore M to its prior state, the same cannot be said about Z whose recovery data must be regarded as
corrupt. How can Z (and other objects at Ni) be restored to a consistent state? A solution to this problem is to provide crash resistant storage (henceforth referred to as a secure storage) facilities such that (consistent) states of objects can be stored on it ('secured'). Appropriate crash recovery procedures can then be introduced for bringing objects back to their secured states. In a distributed system it is also necessary to synchronise the 'securing of objects' such that objects at various nodes remain consistent with respect to each other.

An ingenious protocol (known as the two phase commit protocol) for achieving the above goal has been developed by many workers //2,3,12,13,14/>. The protocol is to ensure that a computation commits only if all of the updated objects have been secured. A familiarity with this protocol will be assumed in the rest of this section. Just as we have taken the view that an object manager is responsible for providing recoverability, it will be assumed that it is also responsible for providing the necessary crash resistance features. We shall discuss some logical details of an interpreter that provides primitive facilities for suitably automating securing of objects.

In the discussion on recoverability, it was assumed that any program at any level i (i>0) can make use of recoverability features provided by l0. Can the same be said about securing of objects? For example, is it meaningful for program Pi (Figure 3) to secure objects used by it and for program Pj not to use secure facility? A little reflection on the reader's part will show that it only makes sense for programs at the highest level of abstraction (such programs are usually referred to as transactions) to use any secure facility. Such a program then makes use of this facility in the manner shown. Before commitment begins, a 'secure' primitive of the interpreter is invoked. The parameter of the primitive specifies the restart point for the process after a crash recovery. Successful execution of this primitive implies that the current states of all of the updated objects have become crash-proof (i.e. can be recreated

```
secure (crash start);[all exceptions: kill]
crash start: discard recovery point;
unlock all the locked objects;
"commit operations"
for all the workers created do [For all exceptions:
  retry the
  delete worker(workernameno, node);
  operation;]
kill(or new());
```

if crashes occur). Once commitment begins, all of the operations must be completed. It is possible that node and communication facility crashes can occur such that these actions take a long - possible infinite - time to complete. So, a distributed system should be sufficiently reliable to reduce the chances of such an event happening to a small probability (see the discussion on the General's Paradox in //3//). As commit operations are unrecoverable, the only
recovery action to take, in case an abnormal condition occurs, is to
retry that operation. Also, any crashes during the execution of
commit operations can imply repeated executions of some of these
operations. These factors must be borne in mind when these operations
(discard, unlock, delete, kill) are implemented. Note that the last
commit action of the executing process - the master process - is
either to kill itself or to execute 'new()' primitive. The 'new'
primitive initialises the calling process to the 'unknown' state (see
below). The process is now ready to execute another application
program.

As a part of the process record of each process, the
interpreter IO at a node maintains a variable 'state' that can take
on the values 'unknown' (initial value) 'secured' or 'committed'. The
'secure' primitive can be implemented by IO as:

```
secure(I) ≡
begin
use calling process . level.recovery data
to construct the set of all object
names recorded;
for all object names in the set do
invoke object name . secure;
with the process record of the calling
process
do {instruction counter := I;
    state := secured }
write the set of object names and
process record on to the secure storage;
end
```

As before, the above is a simple logical organisation that ignores
any efficiency issues. It is assumed that IO has access to some local
secure storage facility which provides atomic read/write operations.

The 'discard' operation of IO is slightly modified as:

```
discard recovery
point ≡
begin
if process record . state = not
committed then "as before"
if process record . state := secured then
{process record . state := committed;
update 'state' on the secure
storage appropriately}
end
```

Finally, a new operation 'crash recover' is provided by IO:
Crash recovery of a node thus consists of invoking the above operation which has the effect of bringing all the objects (that are 'securable', see below) of that node to their latest secured states. Also, processes with state 'secure' or 'committed' are recreated. Crash recover should be a protected operation, not accessible to any object managers.

In order to maintain a 'securable' object, an object manager needs to provide two additional operations - secure and crash recover - that are invoked automatically by IO. Each object manager associates a variable 'object state' with its recovery data; this variable can take on one of the following values: unknown (initial value), secured, or committed. So far we have been assuming that an object manager records, in its recovery data, enough information necessary for the creation of specific prior states. In order to implement the secure operation, it is also necessary for the manager to record enough information for recreating the current state of the object. We assume that this information is recorded as a part of the recovery data (the two parts of recovery data are known as 'undo' and 'redo' parts //3,14//).

Just as an object can be either recoverable or unrecoverable, it is also possible for it to be either 'securable' or 'unsecurable' (i.e. either resistant to crashes or not). Thus, an object can possess any one of the following properties:

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Properties of the Object</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>unrecoverable and unsecurable</td>
<td>object state cannot be automatically restored to that at a recovery point (if any); no crash resistance.</td>
</tr>
<tr>
<td>2</td>
<td>recoverable and unsecurable</td>
<td>object state can be automatically restored to that at a recovery point (if any); no crash resistance.</td>
</tr>
<tr>
<td>3</td>
<td>unrecoverable and securable</td>
<td>object state cannot be automatically restored to that at a recovery point (if any); object state can be made crash resistant.</td>
</tr>
<tr>
<td>4</td>
<td>recoverable and securable</td>
<td>object state can be automatically restored to that at a recovery point (if any); can be made crash resistant.</td>
</tr>
</tbody>
</table>
Clearly, it is desirable, at least for the 'higher level objects' (those that are used in application programs) to be of type 4. In this case therefore, the task of an object manager is, in general, to construct an abstract object that is recoverable and secure out of objects each of which can be of any of the above four types. The disjoint and inclusive recovery schemes discussed in the previous section provide an appropriate framework for formulating and evaluating various possible implementations if these schemes are extended for securability in an obvious manner. However, for the reasons given in the last section, we shall again select the disjoint scheme as the most suitable choice for distributed systems. As an example, consider the object manager M of Figure 4 (assume A, B and C are local objects). Under the disjoint scheme, if process Q executing Pj invokes 'secure()' primitive, then IO will only invoke 'secure' operation of M (even if A and B were securable):

\[
\text{secure} \equiv \\
\begin{align*}
\text{record 'redo' information in recovery} \\
\text{data for objects A, B and C;} \\
\text{object state:=secured;} \\
\text{record recovery data in a secure} \\
\text{storage;}
\end{align*}
\]

Note that the secure storage used by an object manager need not be a local object - it is perfectly possible for a manager to use some remote secure storage. The operation 'crash recover' can be programmed under the disjoint scheme as:

\[
\text{crash recover} \equiv \\
\begin{align*}
\text{update recovery data from that stored} \\
\text{on secure storage;} \\
\text{construct states of A, B and C using} \\
\text{'redo' data;} \\
\text{if object state=committed then object} \\
\text{state:=unknown;}
\end{align*}
\]

The operation discard recovery data will include the following operations:

\[
\text{discard recovery data} \equiv \\
\begin{align*}
\text{delete 'undo' data no longer needed;} \\
\text{if object state=secured then} \\
\{\text{object state:=unknown; make object} \\
\text{state=committed for the secured} \\
\text{recovery data}\}
\end{align*}
\]

One final point to note is that in the disjoint scheme, the interpreter IO must invoke crash recover operations of objects in 'low level to high level object' order. This will ensure, in the example under consideration, that the state of say, A, as constructed after the invocation of the crash recover operation of M, will not be affected even if that operation of A is also invoked by IO.
The case when the concrete representation of an object is distributed over several nodes needs a few words of explanation. As before, assume that M's implementation is distributed as follows: B at Nlocal, A at Nk and C at Nl. Then an operation such as 'secure' would be similarly coded as above except that workers will need to be created at the beginning of the operation (and destroyed at the end). It is easy to see that if Nlocal crashes, the crash recovery procedure for that node will bring A, B and C to their consistent states. But what happens if a node such as Nk crashes? There are four particular cases to consider: (i) Nk crashes (and remains crashed) before the invocation of an operation of M, (ii) Nk crashes during the execution of an operation of M, (iii) Nk crashes - but is 'up' again - during the execution of an operation of M, and (iv) Nk crashes but is up again before the invocation of an operation of M. The first three cases are easy to deal with since a time out exception or process missing exception will be raised when an attempt is made to perform some operation on object A at Nk; the only possible recovery action then is to signal failure to the caller of that operation of M. Case (iv) can be detected only if consistency checks are incorporated in the programs of M. The following structure then suggests itself for all except the crash recover program of an object manager whose concrete representation is distributed:

\[\text{begin}\]
\[
\begin{align*}
\text{create workers at the relevant remote nodes} \\
\text{[for all exceptions: fail]} \\
\text{perform consistency checks on the} \\
\text{relevant remote objects; if OK then} \\
\text{[time out or} \\
\text{crash recover; fail] else} \\
\text{process missing} \\
\text{--normal operations of the program --} \\
\text{exception: fail]}
\end{align*}
\]
\[
\text{end}
\]

Note that if consistency checks detect an error then the crash recover procedure of that manager is explicitly invoked to restore consistency. It is clear that the crash recover program of the manager of M will succeed in restoring consistency of M only if nodes Nk and Nl are up and communication lines are working. Nevertheless, we would like to make the crash recover operation of interpreter IO at Nlocal in no way dependent on these factors (recall that the crash recover program of IO will contain a call on M.crash recover). This problem can be solved if the crash recover programs of object managers with distributed implementations follow the philosophy of "restoring as much consistency as possible". Thus, this program for M can be designed as follows (assuming secure storage is at Nlocal).
crash recover ⋄ create worker (wk,Nk) 
[for all exceptions; goto X]
using secure storage, restore object A at Nk;
X: create worker(wl,Nl)
[for all exceptions; return]
using secure storage, restore object C at Nl;

Assuming that the rest of the programs of M follow the style indicated earlier, eventually M will be restored to consistency. For the sake of simplicity, it has been assumed here that the above programs themselves do not make use of any recovery facilities. Lastly, the scheme discussed towards the end of the last section (for modelling the behaviour of the inclusive scheme) can also be used to exploit the securability, if any, of remote objects. An evaluation of advantages and shortcomings of this approach is left as an exercise to the interested reader.

Operations 'secure' and 'crash recover' for the remote object handler can be programmed as:

secure ⋄

for all the worker names recorded in calling process.level.workerlist do
send message(wl,Nl,secure) fail
receive message( )
calling process.level.workerlist.state=secured;
copy workerlist to a secure storage

-----

crash recover ⋄ create workerlists using the data on secure storage;
"only lists with state=secured are constructed"

The following additional operation is needed in the logic of 'discard' operation presented in the previous section:

discard recovery data ⋄ "as before";
if calling
process.level.workerlist.state=secured
then delete the workerlist;
Finally, crash recovery aspects of a worker process will be described. The modified algorithm for a worker is as shown (the function 'process state' provided by IO returns the state of the calling process). Exception handling during message send and receive operations needs a few words of explanation. If a worker is in a secured or committed state, then it must deliver its response to the just executed command to its - the worker's - master. Therefore, a time out exception during a 'send' results in a retransmission. A process missing exception during a send operation results in the worker destroying itself, unless it is in a committed state; it then has little choice other than to retransmit the message. A similar reasoning applies to the handling of exceptions during a 'receive' operation. The reader is encouraged, at this point, to try out a few crash recovery situations to convince himself that (i) if the master process executing an application program cannot 'secure', then all of the processes created (and still alive) eventually destroy themselves undoing all of their work, and (ii) if the master does 'secure' itself, then it will be able to complete its commit actions (subject to the infinite delay possibility).

"actions of a worker process"

cycle
    wait: receive message ( );
    [time out exception: case process state of
        committed: goto wait;
        secured: prepare 'ok' message;
            goto resume;
        unknown: kill
    end]
    if message = 'establish recovery point' then
        [result:=OK; establish recovery point (resume)]
    if message = 'secure' then secure (wait)
    else {execute the command in the message;
        prepare answer;}
    [for all exceptions: result:=exception type;
        goto resume]
    resume: send message(master,Nmaster,result);
    [time out exception: if process state=committed or
        secured then goto resume
        else kill
    process missing exception: if process state=committed
        then goto resume else kill]
end "cycle"

Since we admit the possibility that the execution of an application program can give rise to a hierarchy of processes of arbitrary depth with master-slave relationships (with one absolute master) as indicated in Figure 1, the relevant algorithms of IO concerned with crash recovery (secure, crash recover, etc) have been designed to treat all processes symmetrically.
CONCLUDING REMARKS

In this paper we have discussed how the abstractions of recoverability and crash resistance can be provided in distributed systems. The approach consists of (i) equipping the underlying machine at each node with some primitive facilities for recovery and crash resistance, and (ii) designing object managers that provide appropriate operations necessary for the maintenance of the above abstractions; these operations are hidden from the users of the objects. In this section some directions for future work will be discussed.

Various issues that are raised when automatic backward error recovery of computations is desired were discussed in the section on recoverability. The crucial assumption made was that necessary to make recovery actions of processes independent from each other, namely, that a process does not access objects that are in uncommitted states because other processes have yet to commit. While this can be quite satisfactory for many applications, there can be situations where the resulting decrease in concurrency can seriously degrade performance. Davies and Bjork's important work on spheres of control //15,16,17// points the way whereby uncommitted objects can be made accessible to other processes in a controlled manner. The control exercised is such that whenever an error is detected, the number of processes that must take recovery actions is known a priori. A different approach has been suggested recently //11// where no such control is exercised; rather, when an error is detected, the set of processes affected is constructed dynamically. It is not clear at present to what a degree these ideas, together with the necessary crash resistance features, should be incorporated in a system. No mention has been made so far as to what happens when recovery is desired after commitment. Any recovery actions undertaken in such a situation are essentially based on an examination of current system state; here again the work of Davies and Bjork on this so-called post-committal recovery provides guidelines for future work. The reader wishing to pursue this subject further may find the review papers //1,8,12// of interest.

No mention was made here about how the recoverability and crash resistance features of object managers can be incorporated in programming languages that support abstract data types. One effort in this direction (for a centralised system) is described in //18,19// where implementation details of a recovery system supporting many of the features mentioned here are also described. Whether the work described there can be extended easily so as to be applicable to distributed systems remains to be seen.

ACKNOWLEDGEMENTS

The work described in the section on recoverability has benefitted greatly from long discussions with P.A. Lee. Acknowledgement is also due to F. Cristian for clarifying certain issues regarding exception handling. This work was partly supported by the Science Research Council (UK).
REFERENCES


//2// B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System". To appear in CACM.


