An Efficient Implementation of Distributed Object Persistence

Norman C. Hutchinson Clinton L. Jeffery

11 July 1989

Abstract

Object persistence has been implemented in many systems by a checkpoint operation in which a snapshot of the object's state is saved to stable storage. This mechanism works well in a single processor system, but in a distributed programming environment it is inadequate. In an application whose components are running on many processors, it is not acceptable for the computation to wait for nodes to recover after a processor failure. The Emerald language checkpoint operation provides a convenient means by which an object may save its state not only to disk but alternately to other nodes on the network. This results in a significant performance improvement in the event of processor failure, especially for interactive distributed applications.

1 Introduction

Local and wide area networks pervade modern computer installations. From the user's perspective, the point of having network access is to provide convenient and even transparent network access to remote resources and data. For the applications programmer wishing to write a distributed application, the network access provided by the operating system can be slow, or difficult to use, or both. While better operating system designs and implementations such as the *x*-kernel, V, and Amoeba systems are solving the former problem, the programmer wishing to write applications for the network is still confronted by extremely complex problems introduced by communications, synchronization, and fault handling in the network.

Within a programming language designed for the single processor model, language support for distributed applications is limited to function call libraries or their equivalents. While this approach has its merits and has met with some success, in most implementations to date it has resulted in well-written, slow distributed applications, or poorly-written, complex network service libraries, or both. More satisfactory performance results have already been obtained in language research projects which integrate the network model into a programming language.

1.1 Emerald

In the Emerald programming language, language primitives move both data- and process-bearing objects from node to node, locate them, and invoke them independent of whether they are local or remote. Emerald is object-oriented in the strong sense that the same object model applies to small local entities such as integers as applies to large global entities which have need of network services. Emerald is a strongly-typed, compiled language. Static analysis allows the compiler to generate very high performance procedure calls for invocations which are local, while generating location- independent remote invocations when the target object may be remote. Further details of the Emerald language, its model, and its implementation have been published elsewhere[?,?].

Recognizing the need for fault tolerance in programs whose parts may be spread throughout the network, the original Emerald design specified object persistence across node crashes in the form of a checkpoint primitive. The checkpoint operation is invoked by an object whenever its state has been modified; in the original implementation the object's image was written to some stable store such as a disk device. Similar mechanisms have been proposed and implemented in many or even most other object-oriented programming environments, and the checkpoint operation's performance characteristics have been thoroughly analyzed. In the case of Emerald, checkpointing an object is very similar to moving it to another node, since in both cases the object must be linearized for transmission, and unpacked before reactivation. While an object is linearized or packed it may be viewed as passive and may not be invoked or otherwise acted upon. After an object is unpacked, any suspended processes are restarted in the case of an object move, while in the case of a recovery after checkpoint, a designated recovery process is started up.

After implementing the checkpoint operation as described above, we found the resulting system functional but inadequate. The remainder of this paper discusses the problem, several solutions which have been proposed in the literature, and the modified semantics we implemented for checkpoint with their resulting performance characteristics.

2 Another Look at the Classic Fault Tolerance Model

The model for machine faults assumed by Emerald is that of the standard Fail-Stop processor[?]. The faults handled are those which are common in practice: a processor fault indicates loss of the contents of main memory on that processor and a complete stop in operation for an undetermined (but generally not infinite) amount of time. Byzantine failure, network partitions, and other critical errors require far more sophisticated mechanisms; protocols which handle these faults are themselves extremely complex distributed applications and make excellent Emerald programming exercises, but they are not provided as builtins.

The language's persistence mechanism is intended, then, to address the relatively common occurrence of a node crash. While checkpointing to disk saves an object's state and ensures forward progress in a computation despite machine failure, the object cannot be reactivated until the disk can be read. In a single processor system this is no inconvenience, but in a distributed environment it is both unneccessary and unacceptable for a large distributed computation to halt indefinitely every time one of the participating nodes crashes. The more nodes involved in a computation the more important it is that the relevant objects remain available.

All distributed computations can be viewed in some sense as interactive applications, since by definition the processes interact with processes on other machines in the network, or with the user, or both. It is this interactive nature which renders a conventional checkpoint operation inadequate in a distributed environment. In the case of Emerald, the programming language mechanisms by which an object was made persistent have been widened to allow that object to remain available not only after a node recovers, but also during the interval between crash and recovery.

The obvious solution to the problem of availability is to create as many copies of the objects as its importance warrants and place them on strategically located nodes in the system. Replicated objects have been proposed in several fault tolerant distributed operating systems, some of which have been implemented[?,?]. In addition to increased *availability* during faults, active replicated objects can provide efficient *access* in many cases such as databases in which many queries are made from different processors to the same data. In applications where read operations are much more frequent that write operations, most operations are local.

Unfortunately, to modify the state of such replicated objects requires a heavyweight combination of communication and synchronization in the form of a transaction or commit protocol. Changing a replicated object's state is both greatly slowed and made more prone to the faults which motivated replication in the first place.

In practical terms for Emerald, such a scheme would have required a major redesign in both the language and the runtime system. Replicated objects in the sense described above constitute a new computational model. In addition to the implementation of transactions, the existing lightweight object motion, location, and remote invocation protocols would have to be replaced by more ponderous counterparts. Transactions are themselves complex distributed applications which make ideal Emerald programming exercises, but correctly implementing their semantics entail more work than many distributed applications require.

Nevertheless, to provide availability despite node faults, it is necessary to replicate information. An alternative to maintaining several active copies of the object whose state must be kept consistent is to maintain several passive copies of the object. Since the passive copies need only be consulted in the rare event that the active instance is lost in a node crash in order to elect a new active instantiation, all of the synchronization costs and most of the communications costs of transactions can be avoided.

Semi-passive replication schemes have been adopted in existing systems such as the Tandem NonStop System and ISIS[?,?]. In most implementations of passive replication to date, the passive copies of the object receive all messages destined for the object (which are broadcast or multicast). They execute the state-modifying operations concurrently with the active copy of the object, but do not send reply messages (aside from whatever is required by the underlying reliable message protocol). These schemes thus avoid certain communications and syncronization costs incurred by active replication. They still incur a large expense in terms of redundant machine cycles as the computation is duplicated for each replica. The more extreme approach adopted by Emerald is to execute the state-modifying operation on a lone active copy and update passive copies via checkpoint upon completion.

Passive copies of Emerald objects originally were created when an object checkpointed to disk. Extending the checkpoint operation to send one or more copies of the object to other nodes required a minimal amount of code in the runtime system and almost no change in the language itself. The existing object distribution protocols continue to concern themselves solely with active object instances; the election protocol comes into play only as a failure handler when the location protocol determines no active instance of the object is available.

3 Language Changes

The Emerald checkpoint primitive has been modified in several ways. The basic operation for most applications will no longer write the object to disk, but rather send out one or more copies of the object to other nodes. For applications requiring a guarantee that an object's state will not be lost, the original disk-based checkpoint operation is available via the new keyword `confirm`. `Checkpoint` and `confirm checkpoint` operations may be freely interspersed. To specify the nodes on which to maintain copies of the object's state, the programmer merely codes

`checkpoint at` $<$ *nodelist* $>$

For the sake of existing Emerald source code, the checkpoint operation in the absence of a preceding `checkpoint at` statement defaults to the original disk-based operation. In order to prevent recovery of obsolete versions of an object, `confirm checkpoint` sends out copies of the object to all nodes on the nodelist for those objects which employ the new `checkpoint at` primitive. The checkpoint-at nodelist is dynamic and may be defined and redefined at runtime. The replication multiplicity, determined by how many nodes you place on your nodelist, is presently limited to a small constant[1]. This number is an exponent in the probability that the object will remain available during any period in which the network is in operation. The particular constant we selected was suggested to us by the literature [Garcia-Molina].

Specifying a nodelist of maximal size will ensure availability except in a severe catastrophe such as network-wide power outage or natural disaster. The simulation studies of Noe and Andreassian have suggested that specifying more than a single passive copy will provide little additional benefit in terms of availability [Noe87]. While this is no doubt true, there are some applications which have extreme availability requirements. Our implementation allows additional copies to be maintained at very little cost other than the space required to store the data.

Object recovery is triggered automatically during remote invocation by failure to locate an active copy of the object in the network. Objects are also restarted when passive copies of unavailable objects are found during garbage collection. The programmer is advised to use the network-based checkpoint operation for all but the most permanent objects, since it is considerably faster during both checkpoint and recovery.

[1]five, in fact

The changes described above retain entirely backward-compatible semantics for existing Emerald applications. To utilize the availability feature most objects which currently checkpoint should be coded with an additional checkpoint-at clause in their initialization. Permanent (i.e. disk-based) objects which also utilitize the added availability feature require the addition of the `confirm` keyword to their existing checkpoint statements. The mechanism is considerably more general than this easy usage suggests, and some object-motion patterns may warrant a more sophisticated replication strategy. The programmer should view replication as another object distribution primitive and use it in concert with the other powerful mechanisms Emerald provides.

3.1 Example

To illustrate the ease and control with which the programmer can now add availability to Emerald applications the facility was added to the Emerald mail system. For those components which were deemed important such as unread message objects, a checkpoint-at clause was added. Once a mail message is read the user may either designate it as important (in which case it remains replicated) or unimportant, in which case the replicated copies are reclaimed and a single copy is written to disk for long term storage. The programmer thus has precise object-level control over which components in an application must remain available during a node crash and which ones, though persistent, the user can temporarily live without. Where appropriate, the programmer can delegate this decision to the user. In many other cases the structure of the data itself suggests to the programmer which objects ought to be replicated. For example, in a filesystem, directories might be replicated so that the name space is preserved even when portions of the actual data are unavailable. This is an instance of the more general truth that it is often appropriate to replicate the internal nodes when objects can be organized in graphs.

In comparison with other mechanisms for providing fault tolerance, our system is extremely easy to use. Unlike systems such as ISIS, Emerald's replication is not entirely invisible to the programmer. The decision of whether, how much, and where to replicate is left to the programmer at the individual object level, without increasing the complexity of the resulting source code.

4 An Implementation of Lightweight Replication

Just as the design change incorporating replication was guided by both the original computational model and the existing implementation, the implementation of replication in Emerald was constructed carefully to fit the existing runtime kernel code. Chronologically, object mobility was the first language feature implementing a linearization of Emerald objects, since it was a primary objective in the distributed systems research of the original Emerald implementors. Disk-based object checkpoint was added later; it required little new code, but the existing linearization code had to be generalized since objects recovering from a crash do not have live processes with activation records as do objects moving from node to node. Passive replication also utilizes the original linearization code written for object mobility. Since passive copies written to disk are identical to those written to the network, no further change to these routines, which constitute roughly 7400 lines of C code, were required.

Support for an efficient election protocol required more substantial changes. The replication node list was added to the object table entry for all global objects. Each checkpoint generates a new sequence number, which is also maintained in the object table. Another field was added to global objects' table entries to contain the pointer to the object's passive image, if any. These changes in the data organization amount to a few bytes added overhead in maintaining the object table. The only other increase in space consumption is that of the replicated copies of objects themselves. On a given node this amounts to no more than doubling the space required to store a single object[2]; since up to five passive copies of an object may be maintained, the network-wide memory consumption could conceivably reach up to six times the amount required in the

[2]keeping a copy on the node at which the active object is located makes sense only for mobile objects. This again suggests that object replication and mobility strategies should be interrelated.

absence of replication. For many applications, a relatively small fraction of the objects (and in particular none of the local objects generated during basic operations) require replication in order to keep the system operational, and there is no reason for most replicated objects to maintain five passive copies. The space requirements are deemed very reasonable in light of the functionality obtained.

The following table suggests that many if not most distributed applications fall into that class of problems in which a small amount of replication in the form of object availability yields a large dividend in overall system functionality during faults.

	Service:	
	Persistence	Persistance and Availability
Application:		
operating systems	data	programs
file systems	data files	(root or major) directories
network protocols	messages	services
...		
graph problems	leaves	interior nodes

As has been noted above, Emerald's location and mobility protocols generally constitute a far greater proportion of the computation than does the checkpoint operation. The first priority of the implementation was to retain the lightweight, high performance nature of these operations. Similarly, within the checkpoint subsystem the checkpoint operation occurs orders of magnitude more frequently than the recovery mechanism. With this in mind, the new replication-based checkpoint system has increased the performance of the checkpoint operation at the expense of a slower recovery phase. In particular, the new mechanism of writing to the network is considerably faster than writing to disk, especially for the common case of a diskless workstation in which writing to disk involved network transmission anyhow.

Passive replicated objects' ability to recover from a node crash long before disk recovery can occur is purchased at the expense of added complexity in the implementation. In addition to the new election protocol for object recovery from the network, the disk-based recovery mechanism required extra logic to abort the recovery in the event that an object replica recovered on another node. Disk-based recovery is already complicated by the fact that objects are mobile and can checkpoint on any available disk. Since disk recovery is infrequent and the number of objects which are actually recovered off disk is greatly reduced in the new system, this slowdown in the disk-based recovery protocol is acceptable.

Network-based recovery involves the election of one of the passive images as a new active copy of the object. Although performance is a first consideration, care is also required to ensure that no obsolete copy of the object is ever recovered, undoing the effects of already completed operations. In order to guarantee this, it is necessary for any election protocol to maintain two properties: (a) a coterie, and (b) disk consistency.

A coterie is simply a list of the sets of nodes which together would have the authority to elect a node to recover the object[?]. Coteries have the property that no two of the sets of nodes in the list have an empty intersection; requiring a coterie simply guarantees that the object cannot be accidentally recovered twice in the event of a network partition. Although a simple majority vote is not always required for a coterie, the combinations of nodes which consistute a majority is a simple example of a viable (but nonoptimal) coterie.

Disk consistency is the property that no copy of an object should be elected if a newer copy of that object has been written to stable store. The simplest way to guarantee this is to synchronously update all the replicated copies of the object before writing to disk.

Emerald's election protocol implements coterie establishment by gaining complete information about the contents of the network. Since an election occurs only as the result of failure in the location protocol, this information is obtained at no extra cost[3]. Location normally consists of an unreliable broadcast query for the object; if no answer is received, the broadcast is followed by point to point queries of each node in

[3]No extra network bandwidth cost. Machine cycle cost is limited to copying 32 additional bytes per location reply message.

the network. In addition to the boolean valued reply to the query for an active copy of the object, each node's reply packet now also includes the sequence number and nodelist of that node's replica, if any. If this information is more recent than that on the querying node, it updates its object table with the newer information. Since the location protocol performs a reliable broadcast, any unsuccessful location attempt guarantees that the querying node will be informed of the most recent passive replica on the network.

Election consists of sending an incarnation order including the sequence number of the desired replica to a selected node[4]. The nodelist is transmitted with each network-based checkpoint image, guaranteeing that all nodes having a copy of the most recent checkpoint will agree as to which node is first on the list. Once an incarnation order is received on that node, it increments its sequence number. Incarnation orders with an old sequence are ignored thus eliminating redundant orders from separate sources. Such situations can occur if two different nodes simultaneously notice that the active instance of some object has become unavailable.

5 Analysis

The mechanism described has been implemented and is running on Emerald networks consisting of suns or vaxen running BSD UNIX. Its performance effect upon very large distributed computations is under testing. On test cases of modest size replicated checkpoint is slightly faster than disk-based checkpoint when no faults occur (insert some figure here). When a fault occurs the effects are dramatic. Lost objects are elected and recovered in less than a second. Since elections take place only when the object is being located or invoked, they take place gradually on demand after a fault rather than all at once.

In order to measure the effect of a replication system on distributed application performance, we present analysis in terms of a variety of resources related to time and space consumption. While the cost incurred by replication depends upon both the caution of the programmer and the number of checkpoints performed, the benefit derived can be expressed as the time required for a given computation multiplied by the machine fault frequency. Short computations are more likely to complete before a machine fault occurs and less expensive to recompute in the event that a failure occurs while they are in progress.

5.1 Availability

We define availability to be the probability that an object will remain continuously available duration an entire computation[5]. In an active replication scheme, or even most passive replication schemes, the availability measure is complicated by the possibility of failure during the processing of a state-modifying operation. The single-active-object model adopted by Emerald corresponds to a more tractable availability index given by

$(1 - F^M)^T$

Where F is the node failure frequency, M is the replication multiplicity or in the case of Emerald the length of the checkpoint-at nodelist, and T is the total time required by the computation.

Availability is a useful measure which appears in several of the resource consumption equations given below. It is the means by which the costs and benefits of replication can be evaluated for these various resource equations. The availability equation presented above also gives the programmer a concrete means of selecting and justifying a particular replication multiplicity based on the particular problem to be solved.

In the examples presented below we present several applications of the above equation with a variety of models. In each case, we assume a hypothetical processor which averages one failure lasting fifteen minutes every two weeks. For the sake of concreteness we select some hypothetical distributed computation with a duration of an hour and a half.

[4] we always choose the first node on the nodelist, which is programmer-specified

[5] This is different from the definition given by [Long89], in which availability is an instantaneous measure and reliability is the continuous measure we call availability.

5.1.1 Example: A Pessimistic Failure Model

Assume for a moment machine failures without recovery[6]. In this case, the probability of a replicated object remaining available for an entire computation is simply the probability that one of the processors will not fail during the entire computation. The effects of replication are very limited and do not correspond to the equation presented above. In the absence of replication this model yields an availability probability of

$(1 - F)^{5400}$

Processor failures are assumed independent. With a single replica the probability that one of the two copies will remain available is

$1 - ((1 - (1 - F)^{5400})^2)$

Extended to five replicas:

$1 - ((1 - (1 - F)^{5400})^6)$

5.1.2 Example: Static Replica Nodelist Assignment

A more realistic model is one in which processors fail and recover. Even if the Emerald programmer is lazy and only specifies a one-time selection of nodes for replication, the probability that the object will remain available is greatly improved. More precisely, failure corresponds not to the probability that all involved processors will go down some time during the computation, but to the probability that all involved processors will be down at the same at time at some point in the computation.

Without replication the probability remains

$A = (1 - .00074^1)^{5400} =$

Given a single replica on the system this becomes

$A = (1 - .00074^2)^{5400} =$

Selecting a maximum replication multiplicity of 5 yields

$A = (1 - .00074^6)^{5400} =$

5.1.3 Example: Dynamic Replica Nodelist Assignment

The above figures assume a trivial replication strategy with a single node list assignment performed during initialization. In reality, it is easy to dynamically reassign the replica nodelist whenever a node crashes[7]. This effectively reduces the chance of unavailability from overlap period (when will M nodes be simultaneously down?) to catastrophe (when will M nodes crash at the same instant?). The effects are comparable to those delivered by Pu's Regeneration Algorithm [Pu].

The effect of dynamic nodelist assignment is to cause the perceived failure frequency to be calculated based on the number of failures rather than the total downtime due to failure. For instance, in the hypothetical situation presented above, coding an application with dynamic nodelist assignment reduces the basic failure probability from .000744 to .0000008. The above equations then become:

Given a single replica

$A = (1 - .0000008^2)^{5400} =$

Selecting a maximum replication multiplicity of 5 yields

$A = (1 - .0000008^6)^{5400} =$

5.2 Computation Delay

Computation delay is a measure of the total time from job submission to completion. Clearly, it is the most important measure in interactive user-oriented applications such as editors and shells. A primary benefit of replication is to bound computation delay due to object unavailability by some small factor dependent upon application size and distribution characteristics as well as the machine failure rate.

[6] An Emerald programmer would have to go out of his way to implement this pessimistic model, but it is useful for comparison.

[7] There are granularity assumptions implicit in this argument, such as the assumption that checkpoints happen more often than node crashes

Since replicated objects lost in a machine failure are recovered as they are needed, applications will be able to proceed long before the system has recovered *all* of the objects which were on the crashed node. In addition, by avoiding disk writes many applications' absolute computation delay are reduced by checkpointing to the network instead of to disk.

5.3 Machine Cycle Consumption

Machine cycle consumption is a measure of the total number of cycles consumed by the computation. The primary checkpoint cycle cost is linearization which is unaffected by switching from disk-based to network-based checkpoint. Replication requires additional cycles on all nodes which receive and maintain copies of other nodes' objects. The election protocol employed by Emerald requires minimal computation. In the event of machine failure, absence of replication requires additional cycles for exception handlers. Most applications do not handle exceptions properly and require restarting and recomputation. In the case of very long computations this cycle cost becomes prohibitive.

5.4 Network Bandwidth Costs

Network bandwidth costs measure consumption of network time, usually expressed in packet counts. Network-based replication requires an allocation of packets proportional to the number of checkpoint operations. Again, in the event of failure, the absence of replication in most current systems results in a great deal of added packets during exception handling and/or recomputation costs.

5.5 Main Memory Requirements

Main memory requirements measure the total number of bytes of main memory utilitized by the computation. Replication increases memory consumption by a small programmer-specified factor for those objects which are replicated, which amount to a small fraction of the number of objects generated by the computation. Intuitively this is the amount of space redundancy required to ensure forward progress in the computation despite infrequent machine failures.

6 Summary

Object persistence in Emerald is a relatively low-level, high-performance form of fault tolerance. It greatly reduces the computation delay incurred during machine faults. Its achieves machine cycle efficiency by choosing passive replication over active or semi-passive replication, avoiding a great deal of synchronization and communications costs during normal computation. Active replication has its merits in terms of improved locality of access, but this service is not always useful and is separable from the objective of fault handling. By gathering election protocol information during Emerald's location protocol, Emerald sends only one additional message per election, obtaining very high network bandwidth efficiency. Replication does require a commitment in terms of replica packets sent and space allocated on participating nodes. The replica packet costs are offset at least in part by the reduction of exception handling packets and recomputation costs incurred in the event of a fault. The space requirements are bounded by some small factor of individual object size and are limited to some small proportion of the objects generated by the distributed computation. The end result is a dramatic improvement in service at a modest cost.