Multi-Model Fault-Tolerant Programming in Distributed Object-Based Systems

Muthusamy Chelliah    Mustaque Ahamad
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280 USA
Phone: (404)-894-2593
(chelliah,mustaq@cc.gatech.edu)
GIT-CC-93/72

Abstract

Extended and replicated transaction models provide consistency and forward progress guarantees for distributed applications that manipulate persistent, shared data. In our earlier work, we developed OS mechanisms that can implement the various transaction models as policies at the user level. The mechanisms facilitate efficiency and flexibility, but programming using the primitive OS mechanisms could be complex. To simplify the programmer's task, in this paper, we develop higher level abstractions. These abstractions can be implemented using the underlying system mechanisms, as part of a language run-time system and library routines. We informally argue the flexibility of our approach for programming transaction models that facilitate nested and colored actions, and Parallel Execution Threads (PET). We quantify the costs of fault tolerance through a prototype implementation of multiple transaction models on the Clouds distributed system.

Key Words: Objects, Transaction Models, Computation Replication Schemes, Distributed Operating Systems, Programming Support.
1 Introduction

Distributed systems have nodes that fail independently. In such an environment, comprehensive fault-tolerant programming should address robustness that preserves data consistency and forward progress which guarantees eventual completion of computations. The atomic transaction model [10] provides robustness for traditional database applications. Computation replication schemes like Parallel Execution Threads (PET) [6] ensure forward progress of atomic (trans)actions. However, the atomic transaction model is considered restrictive for application domains like collaborative work (CSCW) and engineering (CAD and software development) databases. Hence a variety of extended transaction models, that promote more flexible computations like nested [12], colored [18], split [15] and cooperating [9] actions, have been proposed.

We are currently investigating the system support required to build extended and replicated transactions as tools for user-level fault-tolerant programming. The Argus system [11] implemented nested actions at the language level. Alternatively, operating system (OS) support for virtual memory (VM) management, especially a single-level storage abstraction, can be exploited to implement transactions efficiently. The Camelot transactional facility thus implemented nested actions on top of the Mach microkernel [8]. However, there is no one transaction model which can meet the needs of all types of applications [7]. We advocate that an OS should not implement transaction models that correspond to policies for fault tolerance. Hence, we developed earlier [3] OS primitives that can be used to build multiple transaction models efficiently at the user level.

The flexibility offered by parameterized system calls facilitates software reuse of the basic functionality for realizing the different transaction models. Still, integration of transactional recovery control with the VM support provided by the OS ensures that performance goals are not compromised. Nevertheless, programming using the underlying system primitives (i.e., identifying the exact set of system calls and parameters for those calls) to implement a certain transaction model could be non-trivial. The goal of this paper is to develop a programming support module that simplifies the use of the low-level system mechanisms.

A wide spectrum of distributed systems based on language-level support for fault tolerance have been proposed [2]. Object-oriented programming systems, like Arjuna [17] and Avalon [8], provide predefined super classes for recovery and concurrency control from which applications inherit methods for robust programming. FT-SR [16] is a system that extends an existing language,
for supporting multiple paradigms that guarantee forward progress. All these systems employ the nested transaction model as the primary consistency mechanism and hence do not explore abstractions for programming multiple transaction models which can be easily implemented using low-level mechanisms.

To facilitate the implementation of multiple transaction models on the same platform, we develop language-level abstractions to encompass the requirements. The abstractions can be realized using a uniform set of system primitives and run-time data structures which store their parameters. The system primitives can be automatically inserted in the application code with the help of user-defined annotations and a language preprocessor. The programmer can implement many transaction models by changing the attributes associated with the data structures either explicitly or through directives. Thus, the proposed programming support module is simple to use because the OS primitives are made transparent to the programmer, still flexible since the system can be configured by choosing values of parameters for the primitives.

The paper is organized as follows. We first briefly describe the system mechanisms for implementing multiple transaction models. We then show the need for higher level support by arguing that significant amount of information needs to be maintained and manipulated by the user which makes programming complex (Section 2). To reduce this complexity, we identify the minimal amount of information a user has to provide for fault tolerance while specifying an object. The object specification can be automatically converted into a fault-tolerant implementation by inserting system primitives for recovery and concurrency control in the object method code at compile-time. The following main contributions of this paper facilitate this conversion (Section 3):

- language-level abstractions that encapsulate the underlying system primitives,
- run-time data structures for model-specific information manipulated by the primitives,

We illustrate the ease of programming using approach by showing how user-defined annotations result in a fault-tolerant object implementation. We also demonstrate the flexibility of this implementation by accommodating multiple transaction models (Section 4). We have implemented a prototype of the nested transaction model and PET scheme on the Clouds distributed system, to show the viability of our approach and analyze the cost of fault tolerance (Section 5). We conclude the paper with our research contributions and future directions (Section 6).
2 System Mechanisms for Flexible Fault Tolerance

We first describe two transaction models proposed in the literature, so as to highlight their common features. This discussion motivates the system mechanisms which follow, and the higher level abstractions that are developed later. Our goal in this paper is to demonstrate the ease of programming using our system support for robustness as well as forward progress. Hence, we do not show here the mapping of our system support to split and cooperating transactions, that illustrates the flexibility of our approach for realizing a variety of robust applications. However, for completeness, we outline the implementation of colored actions, an extended transaction model, in the appendix.

2.1 Requirements of the Various Transaction Models

In the nested transaction model, an action can acquire a read lock on a data item if all the actions holding the lock in write mode on it are its ancestors. A write lock can be obtained by the action if no action other than its ancestors hold the lock in any mode. Potentially, each action on the path to an executing leaf action can have versions of the various data items. The action must obtain its data version from the nearest ancestor. On completion, locks and data versions created by the action are propagated atomically to the parent on its committal or discarded on its abortion.

Parallel Execution Threads (PET) is a scheme proposed to facilitate forward progress using computation replication [1]. In this scheme, a coordinator action spawns multiple cohorts each of which executes the same action code at different nodes in parallel, unaware of the existence of others. The target action can be completed if one of the cohorts reports successful completion and its changes can be committed. Other cohorts are aborted. A key discerning feature of the PET scheme is that it allows cohorts to share locks since only one of them will be committed at the end.

Thus, the functionality required by the various transaction models is the ability of the actions to create versions of data items and relate them in a model-specific manner by appropriate initialization and committal. Many actions can share the locks on a data item if consistency is not violated. Also, actions should be able to transfer the locks among themselves. In short, an action can obtain data versions and locks from one action and on committal entrust them to another action. We now revisit the system mechanisms, developed by us earlier [3], that satisfy these requirements.
2.2 Programming Paradigm and an Implementation

Objects provide an attractive structuring paradigm in a distributed environment [4]. We predicate our OS mechanisms and language-level abstractions on this popular methodology. We chose the passive object paradigm due to the availability of the object-based system Clouds [6] for experimentation, and lack of efficient and flexible support for fault tolerance in similar systems.

In our system, actions model the active entity in a program - control, and passive objects encapsulate data and code. An action manipulates object data only when it invokes an operation defined by the object. Due to the overheads associated with fault tolerance, we use heavy-weight objects that correspond to shared, persistent, virtual address spaces. Actions in our system are distributed since they cross machine boundaries and address spaces.

Objects are made of a low-level storage abstraction called segment. An OS server implements synchronizers which encompass multi-mode locks used for coordinating concurrent actions executing in an object. Objects store synchronizers as pointers to the information maintained by the server for concurrency control. Next we describe three major functions supported by the system for flexible fault tolerance, and present the respective primitive operations defined on the system abstractions.

2.2.1 Recovery Control

To deal with failures, we allow segments to have multiple versions. Each such version is a distinct segment recognized using a unique system name. Versions can reside on volatile or permanent storage. A segment can be initialized by copying the contents of another segment to it using the clone call. Multiple segments can be atomically committed (copied) to another set of segments using the delegate call. The clone and delegate calls allow robust computations to create data versions and avoid in-place modification. It is however possible to create, manipulate, relate and destroy data versions under user control with these primitives. The operations thus do not dictate a single policy; instead, they facilitate recovery control for implementing many transaction models.

2.2.2 Concurrency Control

An action can obtain a synchronizer using the acquire operation, sometimes even in a conflicting mode with other actions specified as parameters to this operation. The release operation allows
the synchronizers to be transferred between actions. Both these operations can be performed
tentatively if desired; acquired locks need to be discarded when an action implementing a robust
computation fails, and locks are not released till the outcome of the action is determined. The
SyncCommit and SyncAbort operations respectively can be used by a coordinator action to
make such tentative operations performed by a robust action permanent on its success, or void on
its failure. The ResilientSyncBegin call allows the robust action to specify its coordinator.

2.2.3 Invocation

Object invocation is the user-level vehicle for carrying out a computation. An action is created on
an asynchronous object invocation. The Invoke call sets up the VM structures corresponding to
the process representing the invoking action on a certain node. The VM setup is tailored for robust
invocation so as to facilitate creation of data segment versions and their consequent attachment
to the process address space for customized recovery control. To handle failures, we allow a user
action or System - the root action - to monitor the status of any other action through the wait
operation, its parameter being a token that is returned when such actions are created.

2.3 The Need for Effective Programming Support

The two tasks involved in implementing any transaction model using our system mechanisms are:
identifying the exact sequence of required primitives, and initializing and storing the necessary
information for the primitives. The users however may find it cumbersome to insert the relevant
system calls, that manipulate the stored information, in the application code. We next identify
high-level abstractions and develop data structures to realize such abstractions. These abstractions
help automate the insertion of system calls in the user code at compile-time, and initialization of
parameters for the calls, stored in the data structures, at run-time. The resulting transparency of
the system primitives simplifies building fault-tolerant computations.

3 User-defined Fault-tolerant Object Specification

We group the users of our system into three classes: object programmers, action programmers and
application programmers. The former two classes fall under the broad category of system designers
whereas the latter are typically the end users. Multi-model fault-tolerant programming is not entirely transparent, but the required specifications are minimal commensurate with the expertise of each class of programmers.

Our programming support module expects the object programmer desiring fault tolerance to make changes only to the the object specification, not its implementation. The language preprocessor, transparent to the users, can augment the object method code with the appropriate calls to provide concurrency and recovery control specified by the action programmer. The routines that implement a particular transaction model are developed by the action programmers and can be supplied as a library, or object programmers can develop custom implementations of these routines. The application programmer just needs to specify the desired action policy for object invocation.

Since an object invocation represents the basic unit of work in our system, we define it to be the most primitive unit of policy specification for robustness or forward progress. Hence, we need to identify the abstractions which can be used to specify recovery and concurrency control when an invocation is made. We call these abstractions recoverable data and resilient synchronizer. These cannot work in isolation, so we need to demonstrate how to tie them together. We next construct fault-tolerant objects from recoverable data and resilient synchronizers, and show how actions can invoke such objects under the umbrella of multiple transaction models. Only these higher level abstractions, not the underlying system primitives, are made visible to the programmers.

3.1 Interface Description

In Figure 1, we have highlighted the new keywords and class of initialization routines that need to be specified for fault tolerance while defining an object. The object programmer needs to specify the association between the synchronizers and data they guard, through the protects annotation, in the interface definition. We assume programs where each shared data access is protected by a synchronization request. The object method code should contain explicit synchronization calls that protect shared data access. However, initialization and manipulation of the parameters for those calls to achieve transactional concurrency control are transparent to the object programmers.

The action programmer has to provide the class FtPolicy encapsulating the model-specific initialization routines. These routines can access the relevant information stored by related actions to initialize RecSegAttr and ResSynAttr, the data structures storing the attributes of recoverable data
Fault-tolerant Class FtObj {
    // data-lock association defined by the object programmer
    Resilient Synchronizer Sync protects
    Recoverable Data DtItem;
    public:
    method1(argument[1..n]);
    .
    .
    // model-specific routines provided by the action programmer
    Class FtPolicy
    protected:
    init_Coordinator(ResSynAttr*);
    init_Delegator_DelegHoldMode(ResSynAttr*);
    init_Inheritor_InheritMode(ResSynAttr*);
    init_SourceSeg(RecSegAttr*);
    init_DestSeg_RobInvInfo(RecSegAttr*,RobInvInfo*);
}

Figure 1: Fault-tolerant Object Interface

and resilient synchronizers. The routines employ system calls to the action manager, an OS server, for this purpose. We have illustrated in Figure 1 only the simple case where the desired transaction model for object invocation is fixed at compile-time for the entire object. If more dynamic policy control is desired, an application programmer has to specify the attribute DesiredModel as an additional parameter for the object method invocation, or even access of particular data items.

3.2 Higher Level Abstractions and Run-time Data Structures

Now we elaborate the higher-level abstractions corresponding to recoverable data and resilient synchronizer, and the system primitives they encapsulate. We also define the data structures that store the information, manipulated by the primitives, to implement any desired transaction model.

3.2.1 Recoverable Data

Recoverable data readily maps to an abstraction we call recoverable segment. We assume each recoverable data item is stored in a separate segment. Segments that define an object state should be made recoverable, if it is desired that the actions invoking the object execute fault-
struct RecSegAttr{
  // used in segment cloning and delegation
  SysName CurrentSeg, SourceSeg, DestinationSeg; /* segment identifiers */
  VirtualAddress vaddr;
  NumPages size;
}

Figure 2: Recoverable Segment Attributes

tolerant computations. For each recoverable segment, a transient segment is created when an action, conforming to certain transaction model, invokes the object whose data is stored in the segment. The contents of the transient segment are initialized from a source segment storing the object state on data access by the action, and copied to the same or a different destination segment when the action terminates, e.g., a nested action initializes the transient segment from that of the nearest ancestor and copies it to a destination segment held by the parent on commital. Finally, the transient segment is destroyed.

We define a data structure RecSegAttr on per-invocation, per-segment basis as shown in Figure 2 to implement recoverable segments. Figure 3 demonstrates a simple object invocation where the method code accesses data in a single recoverable segment. The key to access RecSegAttr is a 2-tuple consisting of the object header version name ObjVer and segment name Seg. An action can obtain the name of a recoverable segment Seg from the header of the object being invoked.

A transient segment CurrentSeg can be created, cloned from SourceSeg and attached to the invoking process address space so that object data can be manipulated by the invoking action. This simulates working not in-place. On object return, CurrentSeg is detached from the process address space, delegated to DestinationSeg and destroyed. If DestinationSeg is null, the delegation is treated as implicit and without physical data flow. As a result, the action monitoring the invocation inherits the responsibility for CurrentSeg, e.g., a parent action inherits a segment it initially did not have from the child nested action on its commital. Roblnvinfo, a special segment to be described in Section 3.2.3, is used in this implicit delegation.

\footnote{If multiple data items are mapped to the same segment, techniques similar to those adopted in contemporary DSM systems have to be used to deal with problems arising from false sharing.}
```cpp
FtObj::method(RobInvInfo)
{
    SegPtr = RecSegAttr[ObjVer, Seg];// allocate space and return the pointer
    /* RecSegAttr initialized by a model-specific run-time support routine */
    create(SegPtr.(CurrentSeg, size));
    clone(SegPtr.(SourceSeg,CurrentSeg));
    attach(ObjVer,SegPtr.(CurrentSeg, vaddr, size));
    method code
    detach(ObjVer,SegPtr.(CurrentSeg, vaddr, size));
    if(SegPtr.DestinationSeg != NULL)
        delegate(SegPtr.(CurrentSeg,DestinationSeg));
    destroy(SegPtr.CurrentSeg);
    else
        RobInvInfoSeg.SegList[] ++ = Seg;// update RobInvInfo version
}
```

Figure 3: Recoverable Object Invocation

### 3.2.2 Resilient Synchronizer

Resilient synchronizer is the abstraction that provides robust concurrency control. The user can insert **acquire** and **release** operations on such a synchronizer in the object method code in an application-specific manner. Still, the transaction model to be employed for making the application fault-tolerant dictates how these operations are performed in a resilient manner - child nested action's locks are inherited from the ancestors, passed to the parent on successful completion, and discarded on failure.

A data structure **ResSynAttr** defined on per-action, per-synchronizer basis as shown in Figure 4 can be used to maintain the information about a resilient synchronizer. Figure 5 illustrates how a resilient synchronized invocation can be implemented using **ResSynAttr**. An action can designate another action as **Coordinator** for its robust synchronization through the **ResilientSyncBegin** operation. The **Coordinator** action is responsible for committal/abortion of operations on the resilient synchronizer. If the **Coordinator** action itself fails, some action in the ancestral chain leading up to the root action, i.e., **System**, can undo the effects of the acquiring action.

**OverrideTable** stores the condition that needs to be satisfied for an action trying to **acquire** the synchronizer in a certain mode bypassing a predefined conflict rule. Each member of such a
struct ResSynAttr {
    SysName Coordinator, Inheritor; /* action identifiers */
    Mode ReqMode, InheritMode; // enum READ WRITE ANY
    struct {
        SysName Delegator; /* action identifier */
        Mode DelegHoldMode;
    }OverrideTable[MAX-NO-MODES][MAX-NO-COND-ENTRIES];
}

Figure 4: Resilient Synchronizer Attributes

<table>
<thead>
<tr>
<th>Delegator</th>
<th>DelegHoldMode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ancestor[0]</td>
<td>WRITE</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Ancestor[n-1]</td>
<td>WRITE</td>
</tr>
</tbody>
</table>

Table 1: Override Table to Implement Nestion Actions Read Locking Rule

table is a 2-tuple consisting of an action identifier and mode, i.e., Delegator and DelegHoldMode. One typical relation needed to implement many transaction models is that only a specified set of actions, Delegator[1..n](n - nesting depth), can hold the lock in certain modes, DelegHoldMode[1..n]. For example, to implement the read locking rule in the nested transaction model (i.e., an action can obtain a lock in read mode as long as only its ancestors hold the lock in write mode), OverrideTable[read] is shown in Table 1. On object return, the synchronizer can be released to the Inheritor action in InheritMode tentatively, to be made permanent later by the Coordinator action.

3.3 Robust Invocation - Unifying Recoverable Data Management and Resilient Synchronization

Robust invocation is different from the ordinary object invocation supported by the OS. The object header segment has to be cloned so as to facilitate data version creation using recoverable segments. An action now has to keep track of the segments and synchronizers it manipulates and the objects
PtObj::method(RobInvInfo){
    SyncPtr = ResSynAttr[MyId, Sync]; // MyId - current action identifier
    /* ResSynAttr initialized by a model-specific run-time support routine */
    ResilientSyncBegin(Sync, SyncPtr.Coordinator);
    acquire(Sync, SyncPtr.(ReqMode, OverrideTable[ReqMode]);
    method code
    release(Sync, SyncPtr.(Inheritor, InheritMode));
    RobInvInfoSeg.SyncList[++ = Sync; // update RobInvInfo version
    RobInvInfoSeg.MonitoredActIdent = MyId;
}

Figure 5: Resilient Synchronized Object Invocation

struct RobInvInfo{
    Status Completion; // enum Failure Success InProgress
    SysName SyncList[], MonitoredActIdent; // used for robust synchronization
    SysName SegList[]; // keep track of segments delegated implicitly
}

Figure 6: Robust Invocation Information

it invoked. RobInvInfo, shown in Figure 6, helps unify recoverable data management and resilient synchronization. This data structure stores the necessary control information for implementing robust invocations. Completion stores the invocation status. SyncList is the list of synchronizers manipulated as a result of the invocation. MonitoredActIdent is the action executing the robust invocation. SegList contains the list of segments to be inherited by the action monitoring the robust invocation, but not those that are explicitly delegated.

Even though we have orthogonal mechanisms for storage management and synchronization, the results of a computation have to be reflected in a single atomic step, because in the transaction models proposed for fault tolerance, atomicity needs to be maintained. This dictates that system-level object return for robust invocations be specialized so that both concurrency and failure atomicity are achieved through the following events occurring in one step:
1. the data segments already owned by the monitoring action and mapped into its process
   address space are changed, if both the monitoring and monitored actions manipulate the
   same segments,

2. synchronization changes by the monitored action are made permanent relative to the moni-
   toring action,

3. the segments to be inherited by the monitoring action through implicit delegation are added
   to its list of resources.

The atomic step is implemented through the delegation of changed data segments and RoblnvInfo
segment from monitored to monitoring action using a two-phase commit protocol.

The main advantage of the language-level abstractions is the ability to specify inheritance and
delegation for each segment and synchronizer orthogonally. Users can insert these operations in the
application code automatically through annotations, or on their own. The higher level abstractions
are more easy-to-use than the system primitives, still flexible, since they group and encapsulate
similar system calls, and expose only their parameters. Next, we illustrate how to initialize such
parameters for multi-model fault-tolerant programming.

4 Configurable Fault-Tolerant Object Implementation

A fault-tolerant object is comprised of recoverable segments and resilient synchronizers protecting
them from concurrent access. If the segments in an ordinary object are made recoverable, and the
associated synchronizers are resilient, an action invoking the object can easily implement different
transaction models by executing the library routines to initialize RecSegAttr and ResSynAttr data
structures of the various segments and synchronizers manipulated by the invocation.

4.1 Programming Support for Semi-transparent Fault-tolerance

We have shown the generic object code template to be executed for any transaction model in
Figure 7. Robust object invocation subsumes recoverable data management and resilient synchro-
nization. Consequently, the template shown here summarizes the code fragments illustrated in
Figure 3 and Figure 5. For simplicity, we have not shown the object method, system primitives and
// object implementation augmented by the programming support module

FtObj::method()
{
   /* allocate space for RecSegAttr, ResSyncAttr and RobInvInfo */
   
   // prelude
   FtPolicy.initCoordinator();
   FtPolicy.initDelegatorDelegHoldMode();
   Concurrency Control: ResilientSyncBegin and acquire Sync<----user-defined lock
   FtPolicy.initSourceSeg();
   Recovery Control: create, clone and attach Seg version
   
   user-specified method code which accesses Seg
   
   // postlude
   FtPolicy.initInheritorInheritMode();
   Concurrency Control: release Sync<----user-defined unlock
   FtPolicy.initDestinationSeg_RobInvInfo();
   Recovery Control: detach, delegate and destroy Seg and RobInvInfo Versions
}

Figure 7: Augmented Object Method Implementation

run-time support routines with the actual parameters, rather group all the calls into patterns that
achieve recovery and concurrency control with appropriate parameter initialization. User-defined
synchronization calls, lock and unlock, and the protects annotation described earlier collectively
direct the preprocessor to insert the desired system primitives for fault tolerance, and calls to
model-specific parameter initialization routines in the object method code automatically.

Let us consider an example of how the proposed programming support module inserts system
primitives in the application code through user-defined annotations. Association between Sync
and Seg through the protects annotation prompts the preprocessor to insert the prelude to the
object method code that accesses a particular data item after the corresponding lock operation
is detected. The postlude code fragment is inserted once the unlock operation is detected. The
language run-time system allocates arrays of data structures for manipulating the user-specified
recoverable segments and resilient synchronizers contained in the fault-tolerant object. The system
calls then access the data structures to obtain the parameters as described earlier. This func-
tionality accommodates a computation model consisting of many actions, objects, segments and synchronizers, without any programmer intervention other than the object interface specification.

4.2 Accommodating Different Transaction Models

We now describe the fault-tolerant objects that implement different policies for robustness and forward progress. We propose robust invocation of such fault-tolerant objects as the generic methodology for implementing multiple transaction models. We show the flexibility of our approach by realizing two transaction models through appropriate initialization of the various fields of ResSynAttr and RecSegAttr.

An object whose methods are to be invoked as nested actions, AtomicObj, consists of one or more segments, seg[1..n], each protected by a synchronizer of type read/write lock, sync[1..n]. ReplCompObj, an object to be invoked by replicated actions, has two two additional synchronizers Start and Commit. These synchronizers can be used to coordinate spawning and committal of replica cohorts respectively. We have shown in Table 2 the values assigned to the different fields of ResSynAttr of the various synchronizers contained in the fault-tolerant objects implementing multiple transaction models. The corresponding RecSegAttr attributes SourceSeg and DestSeg are owned by the Delegator and Inheritor actions respectively. Many PET cohorts can share the synchronizers Sync[1..n], but not Start/Commit. The cohorts designate the coordinator action ReplCoord responsible for their resilient synchronization.

Thus, it is easier for an action programmer to initialize the various attributes of ResSynAttr and RecSegAttr to realize a particular transaction model rather than manually inserting the relevant system primitives in the object method code, because such initialization pertains verymuch to the locking and recovery rules defined by the action programmers. The object programmer just needs to annotate the object specification appropriately for automatic insertion of the primitives.

4.3 Power of the Abstractions

We now emphasize the virtue of our approach by comparing the recovery control functionality provided by our system with that of the Camelot/Avalon transactional facility [8]. The arguments extend for concurrency control as well as against other similar transactional systems.
<table>
<thead>
<tr>
<th>Model</th>
<th>Nested</th>
<th>PET-Sync[i]</th>
<th>PET-commit/start</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinator</td>
<td>parent</td>
<td>ReplCoord</td>
<td>ReplCoord</td>
</tr>
<tr>
<td>Inheritor</td>
<td>parent</td>
<td>ReplCoord</td>
<td>ReplCoord</td>
</tr>
<tr>
<td>Delegator</td>
<td>nearest ancestor</td>
<td>ReplCoord &amp; ReplCohorts</td>
<td>ReplCoord</td>
</tr>
</tbody>
</table>

Table 2: Resilient Synchronizer Attributes for Multiple Action Models

The primitive segment abstraction in our system just facilitates creating, initializing and atomically committing versions through system calls, i.e., clone and delegate. Consequently, the higher level recoverable segment abstraction requires that the programmer specify only the source and destination segments to be used in such operations. Then, it can use the necessary primitives provided by the underlying segment abstraction for recovery control. Code that computes the parameters for the system primitives to implement common models can be provided by the system designer as run-time library routines. Thus, the end user needs to specify only the data items and associated synchronization variables in an object, and the desired transaction model for object invocation.

In contrast to our recoverable segment abstraction, the atomic super-class in the Avalon system does neither encapsulate version manipulation nor expose version specification. Instead, the whole version management functionality, i.e., manipulation and specification, is encompassed in the underlying Camelot transaction manager. This is in addition to the transparency of the locking rules associated with the synchronization operations. Even though this system is easy-to-program for an end user content with the nested transaction model for robust programming, it is neither flexible for an end user who needs a different transaction model, nor easy-to-program for a system designer implementing multiple transaction models.

The system mechanisms themselves satisfy the flexibility claim of our system support [3]. However, the identification of data versions and related actions holding the synchronization variables to be manipulated by a certain action, possibly spanning multiple nodes, is still complex. This is due to the large semantic gap between the inherently complex target application domain, i.e., multiple transaction models, and the efficient and modular, but primitive system mechanisms. Fortunately, there is a common pattern underlying all the action models which we have exploited to bridge
the above semantic gap through the higher level programming support. The multi-model action programmer now can view the system as a set of actions, and objects containing data and synchronization variables, related by the inherent concurrency and recovery control rules relevant to the specific model. This view is more intuitive than the alternative of a collection of unrelated and parameterized system calls presented by the operating system, e.g., recoverable segment manipulated in a certain fashion vs. primitive segment with all its operations.

The proposed approach thus differs from similar language-level features for fault tolerance:

1. The underlying operating system provides the necessary primitives to implement extended transaction models and computation replication at the user level,

2. The higher level abstractions ease the programmer’s task without sacrificing the flexibility and efficiency offered by the underlying OS.

5 Prototype Implementation

We use the Clouds distributed operating system [6] as the test bed for our prototype implementation. We have demonstrated nested actions and Parallel Execution Threads, both on the same software platform. The hardware consists of a collection of Sun-3/60 and Sparc workstations - connected by a 10 mb/s ethernet - acting as the compute servers and data servers respectively.

5.1 Overview

The various Clouds components - Ra kernel, system servers and Distributed C++ language/runt ime support were already functional when we started our implementation. We had to modify each of these components for fault tolerance support, some times very minimally. We described the changes to the Clouds operating system for adding low-level mechanisms to support fault tolerance in an earlier publication [3]. Here, we concentrate on user-level fault-tolerant programming.

Distributed C++ has library routines which form the gateway into the OS through system calls. The run-time library was enhanced to accommodate the system calls we had modified, e.g., synchronization and action monitoring calls. Storage management calls were integrated with object invocation and were implicitly called whenever robust invocation was requested by the programmer.
We manually inserted the relevant synchronization calls for implementing nested actions and the PET scheme in the object method code. Respective model-specific parameters, i.e., segment and action identifiers, are computed dynamically by the system based on the user directives. All the data structures explained in Section 4 are kept at the system level and manipulated at run-time.

To measure the performance of multiple transaction models, we wrote the class description of a simple object, with an ordinary data variable and a protecting read/write lock. We compiled the class, instantiated the object on one of our data servers, and used a test program which makes the root invocation as a remote procedure call from the user workstation to a Clouds compute server. Root action, i.e., System, is created as a result of this invocation, and spawns a child nested action. In the experiment that measures the cost of the PET scheme, the cohorts were spawned as subactions of a top-level action created by System.

All the numbers reported in Table 3 are measured from the user level. We had to insert hooks in the kernel to obtain the exact profile. The key idea of these measurements is to demonstrate the cost of distributed operations (i.e., those operations which involve more than one node). Each of the entities in our experiments, i.e., object (data and synchronizer) and actions (parent and child in the nested action model, and coordinator and cohorts in the PET scheme), reside on nodes with independent failure modes. Reliable network transmission cost is the major overhead in the implementation of the system mechanisms (null roundtrip of 3.5 ms contributing to approximate costs of 5 ms for synchronization, 15 ms for clone, and 20 ms for delegate of a segment with one 8K page) [3]. We have shown in Table 3 the costs (in milliseconds) of different phases of the nested action and PET scheme implementations.

5.2 Cost of Fault Tolerance

We now analyze our experimental results by investigating how fault tolerance increases the cost of executing a computation. Our main observation is that network communication costs the most for distributed fault-tolerant applications. An asynchronous object invocation entails an action creation message to the remote node in the spawning phase, which results in a page-fault for the object header segment and the overhead of process context setup. The computation phase involves two synchronization requests (acquire and release) and a page-fault request for the data segment. Creator action collects the computation results on its termination through the wait operation.
<table>
<thead>
<tr>
<th>Phase</th>
<th>Ordinary Invocation</th>
<th>Nested Action</th>
<th>PET Computation (2 cohorts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spawning</td>
<td>46.5</td>
<td>46.5</td>
<td>109.5</td>
</tr>
<tr>
<td>Computation</td>
<td>30.1</td>
<td>76.7</td>
<td>137.4</td>
</tr>
<tr>
<td>Termination</td>
<td>4.8</td>
<td>9.9</td>
<td>19.8</td>
</tr>
<tr>
<td>Total</td>
<td>81.4</td>
<td>133.1</td>
<td>266.7</td>
</tr>
</tbody>
</table>

Table 3: Phase-wise Cost of Implementing Multiple Transaction Models (in ms)

A robust invocation (nested action) requires one extra synchronization operation each in the computation (ResilientSyncBegin) and termination (SyncCommit/Abort) phases, and a control message and $n + 1$ page transfers (delegate) for committal at the end of the computation phase. The object is assumed to have $n$ data segments and an invocations status segment, each with a single 8K page, all located on the same site. As shown in Table 3, we have measured this cost at 51.7 ms for a subaction involving an object with a single segment. This is the difference between the numbers reported for a nested action and ordinary object invocation.

Forward progress by computation replication adds to robust object invocation as much remote action creation cost as the desired degree of replication (DR), and 3 synchronization requests for coordinating replicas in the spawning phase (ResilientSyncBegin, Acquire and Release on the synchronizer Start), one page-fault request for the control segment containing the cohort identifiers and 6 synchronization requests in the computation phase (ResilientSyncBegin, Acquire and SyncAbort on the synchronizers Start and Commit) and DR-1 synchronization (SyncAbort) and kill requests each for one unsuccessful PET in the termination phase depending on the degree of replication. The measured cost when DR = 2 is the difference between the elapsed times for executing a 2-PET computation and nested transaction. This is 133.6 ms as shown in Table 3.

### 5.3 Comparison with Related Systems

Now, we relate our experimental results to the ones that have been reported by other relevant systems. This discussion is to show that our system incurs only acceptable overhead despite the flexibility. We give numbers for two kinds of systems, those supporting robust computations, and schemes for replicated computations.
The Arjuna system reports 59.6 msec as a nested subaction cost when the object size is 1K [17]. Rose [14] implemented on a platform consisting of Sun-2 workstations incurs 162.7 ms for committing two large objects each of 4K size. A recent fault-tolerant adaptation of Mach reports 5 transactions per second on a network of Sun 3/60 workstations [13]. As evident from Table 3, we have a subaction cost of 133.1 ms (spawning, computation and termination).

Circus is a replicated procedure call facility implemented as part of the Unix 4.2BSD system [5]. Replicated procedure call with a degree of replication 2 costs 58 msec (real time) and increases linearly with the degree of replication. This cost does not include the creation of a heavy-weight computation at the remote site. This scheme was implemented on a network of VAX-11 workstations connected by ethernet. Fault-tolerant RPC, another replication scheme, incurs an extra overhead of 291 msec per replica [19]. For a 2- replica computation, the cost reported is 593 msec. Sun 3/52M workstations were used as the compute servers. On Sun 3/60s, an active replicated computation (2 PET cohorts) costs 266.7 ms in our implementation, as shown in Table 3.

6 Concluding Remarks

In this paper, we identify higher level abstractions to exploit the underlying system mechanisms. Our approach thus is similar in spirit to existing work for fault tolerance support at the language level. However, the mechanisms themselves are flexible for a system designer implementing multiple transaction models since they obviate reimplementation of the essential features like version manipulation. The higher level programming support module further simplifies the task of multi-model implementation, even for an end user, through automatic insertion of system calls at compile-time and parameter initialization for the system calls at run-time. In the future, we intend to investigate how the new breed of demanding applications like CSCW can exploit our flexible system that implements multiple transaction models. We are also exploring protocols for replicating computations structured using various extended transaction models.

References


A Colored Actions

Researchers desiring even more flexibility than nested atomic actions have developed models that facilitate **serializing** and **glued** actions [18]. A serializing action $T_s$ acts as a top-level action for an enclosed action $T_1$ from the point of view of concurrency atomicity, but not failure atomicity. This property can be used to transfer some locks owned by $T_1$ to another action $T_2$, also enclosed within $T_s$, on $T_1$'s completion. This behavior minimizes the effects of failures on long-lived actions, because changes made by an action to data items can be made permanent periodically without releasing the locks. Gluing action $T_g$ is used to transfer some modified data items as well as associated locks, but not all, from $T_1$ to $T_2$. This behavior releases unwanted resources dynamically, and facilitates long-lived, but highly concurrent, applications.

**Colored** actions, which associate multiple colors with data items, locks and actions, provide a unifying base to implement such models which extend, but not relax, serializability. Locking rules for colored actions are very similar to that of nested actions. The distinction is the ability of the actions to transfer their resources, data and synchronization variables, on committal to more than one ancestor atomically depending on the associated colors. A fault-tolerant object, **MultiColoredObj**, facilitating colored actions, has segments and synchronizers that can be accessed using different colors at run-time. A multi-colored action inherits a certain colored segment and synchronizer from the nearest ancestor with the same color, and delegates the segment and resilient synchronization responsibility to the same action on committal. The colored action programmer can accordingly initialize the various attributes of **RecSegAttr** and **ResSynAttr** as described earlier.