Linking Consistency with Object/Thread Semantics: An Approach to Robust Computation[1]

Raymond C. Chen
(chen@sni-usa.com)
Partha Dasgupta
(partha@enuxva.eas.asu.edu)

[1]This work was supported in part by NSF contract CCR-86-19886 and NASA contract NAG-1-430. This paper appeared in the *Proceedings of the Ninth International Conference on Distributed Computing Systems* (IC-DCS), June 1989.

Abstract

This paper presents an object/thread based paradigm that links data consistency with object/thread semantics. The paradigm can be used to achieve a wide range of consistency semantics from strict atomic transactions to standard process semantics. The paradigm supports three types of data consistency. Object programmers indicate the type of consistency desired on a per-operation basis and the system performs automatic concurrency control and recovery management to ensure that those consistency requirements are met. This allows programmers to customize consistency and recovery on a per-application basis without having to supply complicated, custom recovery management schemes.

The paradigm allows robust and non-robust computation to operate concurrently on the same data in a well-defined manner. The operating system need support only one vehicle of computation — the thread.

1 Introduction

The ability of a system to maintain consistent data in the face of hardware breakdowns, computation aborts, and other forms of failure, is loosely termed *robustness*. Robust computation transforms data from one consistent state to another in spite of failures, where the definition of consistent is application dependent. Thus robust systems may be termed "consistency-preserving" systems. In a general purpose (distributed) system, there is a need for both robust as well as non-robust computations and data, as many applications do not want to (or have to) pay the overhead costs needed in managing robust computations.

The most popular paradigm for robust computation are the *atomic transaction* based paradigms. The transaction paradigm is based on the *totality* concept — in principle, all the effects of a transaction are reflected in the *stable* (or *permanent*) state or none are. However, the transaction paradigm can be quite restrictive and most systems that provide transactions ([All83] [LS83] [PN85] [SBD*84]) also provide "escape" mechanisms by which users may take advantage of application-specific semantics to increase concurrency and decrease overhead without sacrificing correctness. For example, *Argus* ([LDH*87]) allows atomic actions to touch non-atomic data, thus allowing actions to communicate. A paradigm developed for *Clouds v.1* provides recoverable and non-recoverable data segments, as well as custom locking and commit handling ([AM83] [Wil87]). These can be tailored for a variety of applications that cannot run as strict transactions. The *Avalon/Camelot* system ([HW87]) allows users to write their own commit routines to provide custom recovery for similar purposes. *Quicksilver* ([HMSC88]) goes further, allowing customized commit protocols as well as commit and recovery routines. However, writing algorithms that correctly use these custom locking and recovery schemes is left to the application programmers. These algorithms can be quite complex and intricate. *Locus* ([WPLP85]) allows processes and actions to co-exist. However, a transaction locks the portions of the files it accesses and the locks apply to both processes and transactions. This preserves serializability for transactions but penalizes processes when a transaction is accessing the same data.

We have developed and are implementing a paradigm that we feel meets the following goals:

- Allows the object programmer to tailor consistency and concurrency in a manner appropriate to each application.

- Supports a range of consistency requirements from best-effort consistency to strict atomic transactions.

- Enables the operating system to handle concurrency and consistency issues in a transparent manner, without requiring the programmer to develop complex locking and recovery schemes tailored for each application.

- Presents one uniform world-view that encompasses both robust and non-robust computation, allows them to co-exist in the same system and if desired, concurrently access the same data with well-defined results.

This paradigm is designed for systems supporting an object/thread model of computation ([WCC*74] [Jon79] [Lis82] [LS83] [All83] [ABLN85] [BMS85] [B*85]). The only vehicle of computation in this paradigm is the thread. Consistency and concurrency control are expressed through the semantics of objects, object invocation, and thread creation.

The paradigm is being implemented as part of the *Clouds v.2* operating system, a passive object-based general-purpose distributed system. Clouds v.2 is built around the *Ra* kernel, an extendible, minimal kernel that provides light-weight processes, segment-based virtual memory management, short-term scheduling, and the ability to plug in system-level objects to perform additional operating system level functions ([BAHK*89]). Ra has been implemented on Sun 3/60 workstations, is operational and the Clouds v.2 implementation effort is currently underway.

Section 2 of this paper presents the basic object/thread model. Sections 3-6 present the paradigm while sections 6 and 7 deal with issues arising due to concurrent execution.

2 Objects and Threads

An object is a long lived entity containing state (persistent data) and a set of operations that operate on this state. The object can be thought of as a virtual address space that is named and permanent. In fact, objects can be implemented as permanent virtual spaces ([DLA88]). This virtual space contains the data as well as the operators. The operations are called *entry-points* to the object.

Operations on objects are invoked by threads of execution (or *threads* for short). A *thread* is the carrier of execution in a distributed system much like a process is the carrier of execution on a centralized system. The entry of a thread into an object space is called *object invocation*. Object invocation allows parameters to be passed to the operation being invoked at the discretion of the object programmer. A thread begins its computation at an entry point in an object and flows though all objects (if any) invoked by this operation. Invoked objects may reside on any site in the distributed system. The object invocation terminates when the operation *completes* (returns). Note that an operation on an object can invoke operations on other objects (or operations on the same object). Objects are *passive* in the sense that they define the data and the operations, but do not execute by themselves. The threads are the active component of this programming model. This programming model is based on the processing environment provided by the *Clouds* operating system.

3 The Basics

Our approach is based on the notion of three types of consistency that an object programmer would want to maintain — global consistency, local consistency, and standard. Global consistency reflects a need on part of the object programmer to maintain a group of cooperating objects in a consistent manner. Properly used, global consistency guarantees that the operating system will automatically control concurrency and recovery across a set of objects so that their permanent states will stay both internally consistent and consistent with respect to each other.

Local consistency, on the other hand, is appropriate for those circumstances where *inter*-object consistency may be too strong a criterion. Sometimes, consistency within one object, or *intra*-object consistency is enough. Local consistency, properly used, guarantees that the system will control concurrency and recovery in a way that the object state will always stay internally consistent.

The standard degree of consistency is the degree of consistency that users in the process world have become accustomed to — that is to say no guarantees at all. If nothing fails, the data will be consistent, but the system can not guarantee consistency in the face of failure.

Object programmers label object operations. These labels indicate to the operating system the type of consistency desired by the object programmer for each operation. Threads carry consistency labels which indicate the type of consistency the thread currently supports. When a thread invokes an object, the thread and operation labels are checked to see if they are compatible. If they are not, the thread *transforms* to meet the requirements of the object operation. The thread transforms back to its previous state when the operation terminates, whereupon the system performs the appropriate synchronization and recovery processing.

Objects contain persistent data items and reside in a single-level store backed by stable storage ([LS79]). The versions residing in the single-level store are known as the *base* (or *permanent*) *versions* while those residing on stable storage are known as the *stable versions*. Stable versions are presumed to survive system failures. No such assumption is made about the base versions.

4 Consistent Operations

Each entry-point is marked with one of four labels by the object programmer:

- Global Consistency Preserving (GCP)
- Local Consistency Preserving (LCP)
- Standard (S)
- Inherited (I)

Each thread of execution in the system also bears a consistency label. A thread may be a *global consistency preserving* thread (*gcp-thread*), *local consistency preserving* thread (*lcp-thread*), or a *standard* thread (*s-thread*) depending on whether the state of the thread's consistency label is set to global consistency preserving, local consistency preserving, or standard at that time. Both gcp-threads and lcp-threads reflect a commitment towards at least a minimal degree of data consistency and can be classified as different types of consistency-preserving threads (*cp-threads*). Thus in cases where the discussion applies to both lcp-threads and gcp-threads, we shall refer to both gcp-threads and lcp-threads as cp-threads and distinguish between them only when necessary.

When a thread with consistency label X invokes an operation T marked with consistency label Y, if X is not *compatible* with Y, then the thread *transforms* to a thread bearing

Thread Label at invocation	*Operation Consistency Label*			
	GCP	LCP	S	I
GCP	No change	T $\rightarrow$ LCP	T $\rightarrow$ S	No change
LCP	T $\rightarrow$ GCP	T $\rightarrow$ LCP†	T $\rightarrow$ S	T $\rightarrow$ LCP†
S	T $\rightarrow$ GCP	T $\rightarrow$ LCP	No change	No change

Note: "T $\rightarrow$" $\equiv$ "Thread transforms to"
† Though the label remains the same, this is a transforming invocation.

Figure 1: Consistency Label Compatibility

consistency label Y. The consistency label of the thread reverts to the original value X when operation T completes. This process is called *thread transformation* and T is termed a *transforming invocation*. Thread label/operation label compatibility is summarized in figure 1.

4.1 Global Consistency Preserving

If a thread invokes a *global consistency preserving* (gcp) operation and the invoking thread is not a gcp-thread, the thread becomes a gcp-thread and the current invocation is a *transforming invocation* for that thread.

When a gcp-thread attempts to update a persistent data item, the update is not immediately reflected in the permanent version of the data item. Instead, a new *shadow* version of that item is created. Further updates to that item by the same thread are reflected in the shadow version.

Let a *change-set* be the set of the names of all data items for which a gcp-thread has created a shadow version. A change-set is associated with each gcp transforming invocation. A data item is *named* by a change-set if the name of the data item is contained in the change-set. Every time a gcp-thread updates a persistent data item if the item is not named by the change-set of the transforming invocation for that gcp-thread, the name of the item being updated is added to that change-set. Upon completion of the transforming invocation, the change-set for that invocation will be atomically committed (or aborted) to the base and stable versions. A change-set is committed by atomically setting the values contained in both the base and stable versions of all items named by the change-set equal to the values contained in the shadow versions of all items named by the change-set. This normally requires executing a 2-phase commit protocol ([Gra79]). We refer to this process as committing changes or committing a change-set. A change-set is aborted by simply discarding the shadow versions of all data items named by the change-set.

If a gcp-thread invokes a gcp entry-point, the thread remains a gcp-thread and the operation is executed. Since the thread was already a gcp-thread when the current operation was invoked, the thread must have already invoked a transforming invocation. Therefore, updates made during the current operation are not committed when the current operation completes. The updates are reflected in the change-set of the transforming invocation and are committed (or aborted) when the transforming invocation commits (or aborts) its change-set. Since gcp-threads operate on shadow versions, no change made while the thread was a gcp-thread will be made permanent unless the transforming invocation successfully completes and all shadow versions are successfully committed.

4.2 Local Consistency Preserving

Local consistency preserving (lcp) operations are similar to gcp operations in that all changes made during an lcp invocation are made on new shadow versions and atomically committed (or aborted) when the lcp operation completes. However, *every* invocation of an lcp operation is a transforming invocation that transforms the invoking thread into an lcp-thread regardless of the previous state of the thread. That is, all changes made in an lcp invocation are committed or aborted when the invocation terminates even though the thread may have been executing as a gcp-thread before invoking the lcp operation.

Like gcp operations, each lcp invocation has a change-set associated with it and that change-set is committed (or aborted) when the invocation completes.

4.3 Standard

If a thread invoking a *standard* (s) operation is not an s-thread, then the thread is transformed into an s-thread and the current invocation is a transforming invocation. Upon completing the invocation, the thread reverts to its previous state.

Standard invocations do not commit or abort their changes, thus they do not have a change-set associated with them. If a thread updates the value of a data item in a standard invocation, that update does not create a shadow version. The update is instead applied directly to the latest version of the data item. The system will eventually propagate the changes to stable storage, however, the system makes no guarantees as to when this will occur. This is consistent with the "best-effort" semantics of s-threads and the notion of a process. (The definition of "latest version" will be discussed in greater detail in section 6.3.)

4.4 Inherited

Inherited operations are compatible with all threads and indicate the absence of a consistency requirement. Thus, inherited entry-points inherit the consistency label of the entering thread. If the invoking thread is a gcp-thread, the invocation behaves as a gcp invocation would. The same is true of s-threads or lcp-threads.

Inherited operations can be used to provide the object programmer with even more control over object consistency. Inherited operations can be used for operations that do not access persistent data or as "filters". Inherited entry-points provide a means by which objects may first examine the consistency label on the thread before allowing further processing to occur. Depending on the state of the thread and the object implementation, the object may allow the invocation to proceed (possibly invoking an internal entry-point to perform the actual processing) or it may reject the invocation by returning to the caller. This allows an object programmer to provide a uniform external interface for all threads while using different implementations for different thread types (gcp, lcp, or s).

5 Thread Creation

We allow threads to create other threads to execute object operations. The newly-created thread is said to be the *child* of the thread that created it (which in turn is referred to as the *parent* thread). A parent thread may monitor the status of its children, kill, suspend, and resume them. All threads execute concurrently.

In addition to consistency labels, all threads are also assigned a *nesting* attribute. Threads may be either *top-level* threads or *nested* threads. If the parent thread is a gcp-thread, the child thread may be created as either a top-level thread or a nested thread. Otherwise the child is created as a top-level thread.

Top-level threads are s-threads when first created. Nested threads are gcp-threads when created and the first operation invoked by the thread is a transforming invocation for that thread. If a nested gcp-thread transforms into a non-gcp thread, it is treated as a top-level thread until the invocation that transformed the thread into a non-gcp thread terminates.

Top-level threads commit their changes to permanent state. Nested threads commit their changes to their parent. That is, changes committed by a nested gcp-thread are not immediately reflected in the permanent system state. Instead, those changes are passed onto the parent. This will be discussed in greater detail in section 6.

6 Locking and Visibility

Visibility becomes an issue when two or more threads concurrently executing within the same object attempt to read or write the same data item. Consider the read or write a *touch* on that item. In our paradigm, the operating system enforces certain system level locking rules on permanent data by requesting and releasing system-level locks on behalf of some threads that attempt to touch that data. These locking requests and the locks themselves are neither visible nor controllable by the user. This locking is 2-phase ([EGLT76]) by definition as will become obvious in the discussion below and is performed automatically by the operating system.

6.1 Consistency-Preserving Threads

Consistency-preserving threads automatically lock any data they touch. Read/Write locks are automatically requested by the operating system on behalf of the executing cp-thread when the thread attempts to read/write a data item.

The locking rules for interacting cp-threads are similar to the ones described in [Mos81] for nested actions. Thread creation may lead to a tree of one or more cp-threads, all with a common top-level ancestor. We call these thread trees consistency-preserving thread-trees.

Let T be a cp-thread in a cp-thread tree. A thread U is an ancestor of T if it lies on the path from T to the root of the thread-tree. This path includes T itself, thus a thread is always an ancestor of itself. A cp-thread T has a depth of n if and only if the path from T to the root of the tree is n nodes long where the length of the trivial path (a node to itself) is length 1. A *live* cp-thread is a non-terminated thread that has invoked but not yet completed at least one transforming consistency-preserving invocation. A cp-thread T

may write-lock permanent data only if all other live cp-threads that hold a read or write-lock on that data are ancestors of T. A cp-thread T may read-lock permanent data only if all other live cp-threads that hold a write-lock on that data are also ancestors of T. A cp-thread can not read (or write) a data item unless it obtains a read-lock (or write-lock) on that item. If a cp-thread attempts to read (or write) a permanent data item and the system can not grant that thread the read (or write) lock due to the locking rules, the thread blocks until the lock can be granted.

In addition, to prevent undesirable interactions between concurrently executing parents and children on a cp-thread tree, a thread may update permanent state only if it has no live descendants. This constraint ensures that all non-terminated children of a thread will see the same (intermediate) versions of that thread's changes (if any) to permanent state.

Any locks held by a nested cp-thread are propagated to its parent if the thread successfully commits or released if the thread aborts. (Aborts are presumed to return only a failure code to the caller.) Locks held by top-level cp-threads are released when the transforming invocation commits or aborts. These are the *only* situations where system-level locks are released. Aborting a cp-thread in a thread-tree automatically aborts all its descendants.

6.2 Version Stacks

The cp-thread locking rules outlined in section 6.1 are a straight-forward adaptation of the rules laid out for nested actions in [Mos81]. These rules lead to modeling uncommitted changes to a data item by versions on a *version stack*.

Every persistent data item can be viewed as possessing a version stack with the permanent version at the base. The entire stack exists in volatile memory, however, the permanent versions are backed by *stable versions* residing on stable storage and all commits to permanent versions are mirrored on the stable versions.

The versions on the version stack are referred to as versions $0, 1, 2, ...n-1, n$ where version 0 is the permanent version, version n is the top version and n is the height (or depth) of the stack. Version creation for gcp-threads is based on the nesting level of the thread in the thread-tree. Version creation for lcp-threads is based on the number of times different unfinished lcp invocations in the same object have touched the same item. In both cases, the idea behind the version management is to allow gcp and lcp invocations to operate on private shadow versions and then atomically commit or abort those versions.

When a gcp-thread T of depth d gains a write-lock on a data item it has not previously touched, a new version of that item (call it X) is created and initialized to the value of the version on top of the version stack. Let n be the depth of the version stack. If $d - n > 1$, then $d - n - 1$ new versions are also created, initialized to the value of the top version on the stack, and placed onto the version stack as versions $n + 1, n + 2, ...d - 1$. X is then placed on top of the stack as version d. If $d - n = 1$, X is simply placed on top of the version stack as version d.

When an lcp-thread touches a data item it has not touched in the current lcp invocation, a new version is created, initialized to the value of the top-most version on the version stack, and placed on the stack.

The locking rules together with the rule that a thread may update permanent state only if it has no live descendants ensure that there exists only one version at each level of the stack. Thus the stack is always a *linear stack* and never a *cactus stack*.

When a cp-thread commits, all items in its change-set are committed or aborted. A gcp-thread of depth n commits a change by copying the top version of the item on the version stack (version n) to the next-most-recent version on the version stack (version $n-1$) and then discarding the top version. Thus nested gcp-threads commit their changes to intermediate versions that will then be committed or aborted by their parents. However, having a depth of one, top-level gcp-threads commit their changes to the permanent versions.

A gcp-thread tree may create a version stack with a depth of greater than one if non-root threads update data. An lcp-thread may also create a version stack with a depth of greater than one through recursive invocations. However the semantics of local consistency demand that the effects of an lcp invocation be permanent if the invocation completes successfully. Therefore, an lcp-thread commits a change by copying the top version on the stack to all other versions on that stack and then discarding the top version. In effect, an lcp-thread commit is a *write-through* commit.

All cp-threads abort a change to an item by discarding the top version of the version stack for that item.

6.3 Views

The locking rules in section 6.1 are designed to allow only one cp-thread to hold a write-lock on a data item and update it without blocking. However, these updates affects only the top-most version on the version stack. It follows that for cp-threads, the top-most version on the stack can be thought of as the *latest* version since it contains the latest update and is the only version that can be read or written by a cp-thread.

s-threads, however, may operate directly on the permanent version of a data item. Assuming for the moment that no cp-thread is working with data items used by s-threads, the permanent versions being updated by s-threads are also the top-most versions on the version stacks for those items.

This leads to two appealing global views of the system. One is the *stable view* consisting of the stable versions of the persistent state. This view would be present even if the system were to crash that instant and then recover. Another view is the *instantaneous view*. This view consists of the latest version of each persistent data item, which corresponds to the top version on the version stack for every persistent data item whether that version be a shadow version or a permanent version. This instantaneous view may not be valid in the case of failures. However, it is an intuitively appealing view in that, at the instant the system is examined, it is an accurate snapshot of the latest changes made to the system (see figure 2).

7 Thread Interactions

In our paradigm, s-threads always operate on the instantaneous view. Changes made by s-threads become immediately visible to all other s-threads since the change is reflected in

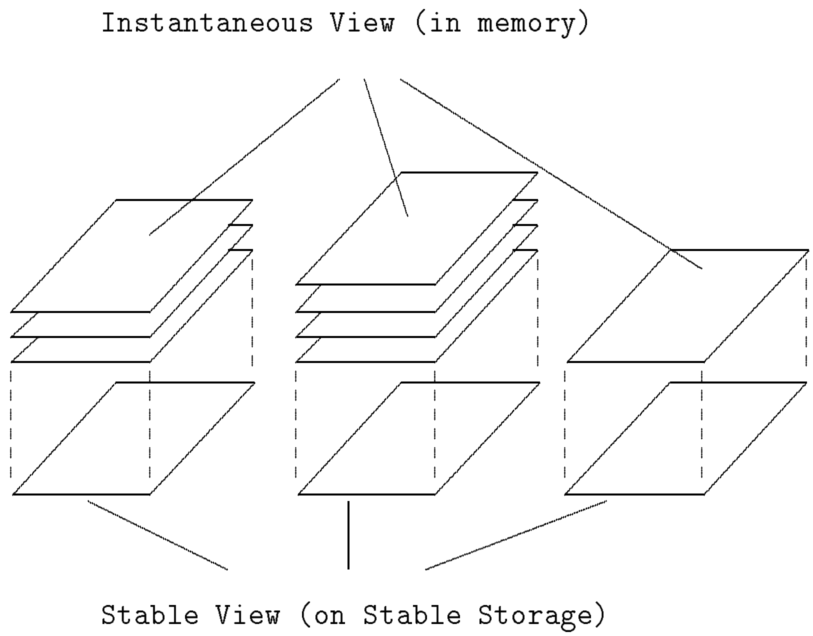

Figure 2: Instantaneous and Stable Views

the instantaneous view. If the changes are made to a permanent version, barring a system failure, those changes will eventually be propagated to the stable view. This is consistent with the usual process semantics.

cp-thread trees operate on the instantaneous view *and* alter the stable view by committing portions of the instantaneous view to the stable view. This allows programmers to control consistency in the stable view and is consistent with the usual notion of a transaction. The change-set of a transforming invocation for a top-level cp-thread at commit time can be regarded as the set of data items that have to be committed to bring the stable view into agreement with the portion of the instantaneous view that has been altered as a result of that invocation. However, while the locking and visibility rules defined in section 6.1 regulate the interactions between cp-threads, they do not address the situations that may arise when s-threads and cp-threads attempt to operate concurrently on the same persistent data.

For in our paradigm, standard threads do not acquire system-level locks. We feel that this would be too restrictive. Forcing s-threads to acquire system-level locks would be treating them like *de facto* transactions. This seems redundant given that a programmer can prevent s-threads from interacting with cp-threads inside an object by using inherited entry-points or lcp/gcp operation labels. If the programmer wishes to use s-threads in an object and synchronize their activity, user-level locking or some other form of explicit concurrency control may be used. This is again consistent with the process paradigm. However, this lack of full automatic synchronization leads to three situations that must be defined: s-threads reading writes made by cp-thread trees, cp-threads reading writes made by s-threads, and s-threads overwriting updates made by cp-thread trees.

7.1 Standard Reads of Consistent Writes

s-threads do not acquire system-level locks, hence they will never be blocked by the system when accessing permanent state. Since s-threads see the instantaneous view of the system, they see the latest versions of all data *including* changes made, but not committed, by cp-threads. Thus, if an s-thread attempts to read data touched by a cp-thread, the s-thread will not block and will read the latest (instantaneous) version of the data.

7.2 Consistent Reads of Standard Writes

S-threads may write to data that have been read-locked by a cp-thread. The write will update the latest (instantaneous) version of the data. A cp-thread commit or abort affects only versions of data items *created* by that cp-thread and its committed descendants and placed on version stacks. Reads by a cp-thread do not create new versions on the version stack. If a cp-thread and/or its committed descendants read a data item updated by an s-thread and neither the cp-thread nor any of its descendants updates that item, the commit processing for that cp-thread will not include the update made by the s-thread.

7.3 Standard Writes and Consistent Writes

We allow s-threads to overwrite uncommitted updates made by cp-threads. If this happens, the s-thread overwrite does not invalidate the system-level locks on the overwritten data. The overwrite appears in the instantaneous view and will be committed or aborted when that item is committed or aborted by the concurrently executing cp-thread *exactly* as if the update had been made by the cp-thread. Thus, in this case, the permanence of the s-thread overwrite is in doubt and is governed by the commit processing of the cp-thread holding the system-level write-lock on the overwritten data item. This, however is consistent with the "best-effort" semantics of s-threads.

7.4 Mutable Threads

Figure 3 shows an example of a gcp-thread invoking a number of object operations. Operation A (in object #1) is a gcp entry-point. Operation B (in object #2) is an lcp entry-point. Operations C and D (in object #3 and #2 respectively) are standard entry-points. In this example, all the invocations except the invocation of operation A (or invocation A) are transforming invocations. The changes made in segments #1, #5, and #7 while the thread is a gcp-thread are visible in the instantaneous view as soon as they are made. However, the affected items are system-locked and the changes will not be committed or aborted until the transforming invocation (not shown) that made the thread a gcp-thread completes. Changes made in thread segment #2 and #4 are simultaneously committed to the stable view or aborted when invocation B completes. Changes made in thread segment #3 (invocation C) are performed without system-locking. Thus, those changes are immediately visible in the instantaneous view regardless of the state of invocation B. They may eventually be committed to the stable view by a cp-thread or committed by the system or aborted. However they will not be committed or aborted as a direct result of any of

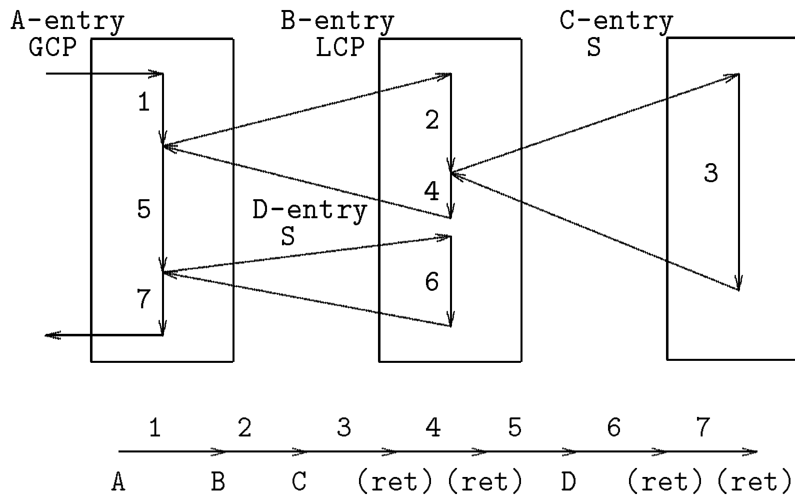

Figure 3: A Thread of Execution

the invocations shown in figure 3. Likewise, changes made in thread segment #6 become visible in the instantaneous view without affecting system-level locks and the changes will eventually be committed or aborted (but not as a direct result of any of the invocations shown in figure 3).

8 Comments

Given this set of thread creation and consistency semantics, we can support the functionality of top-level and nested actions. If a standard thread invokes a gcp operation, the thread is transformed into a gcp-thread for the duration of the object invocation and becomes the equivalent of a top-level action until the operation completes or invokes an lcp or standard operation. A top-level action can be also created by creating a top-level thread to invoke a gcp operation. A nested action can be created by creating a nested thread to invoke a gcp object operation.

If a set of objects are programmed so that only cp-threads are allowed to execute object operations, all operations will be serializable and the above-mentioned actions will behave as strict atomic transactions. If the objects are programmed so that only cp-threads may update permanent data or so that cp-threads do not read updates made by s-threads (if any), then every "action" (cp-thread tree) will be serializable with respect to every other cp-thread tree.

Serializability breaks down only if s-threads are allowed to view a cp-thread's intermediate results *and* make their own results visible to other cp-threads. However, each object has full control over the thread interactions that may occur within it. Although interactions between s-threads and cp-threads are not controlled by the system, they are under the control of the object programmer on a per-object basis.

Thus this paradigm allows for consistent updates to permanent object state in accordance with the view/failure atomicity paradigm and nested action semantics per [Mos81]. Interesting and complex semantics may also be achieved by using well-defined system properties that in other systems would have to be programmed in a completely custom manner. The system properties may also be supplemented by using user-level concurrency control.

9 A Robust Object File System

Robust file objects on a disk system can be defined as in figure 4. These robust file objects may be read from and written to. The *read* and *write* operations are lcp but appear to behave like atomic reads and writes to users of the object. Reads will always read consistent data and writes will not leave the object in an inconsistent state. The *truncate* operation (not shown) truncates the file at a specified block. The *segment* directives form system-locking segments (see section 10). Parameters are assumed to be in/out by value.

The data in a file object are contained in a number of fixed size "virtual" blocks. Before being used to hold data, virtual blocks must be backed by real storage blocks allocated by the disk block allocator. If a file needs to grow (caused by a write at the end of the file), all file objects call the disk block allocator object. The block allocator manages a disk map, allocation statistics, and keeps track of which object has been allocated what blocks. This information is all bundled into one system-locking segment.

The block allocator has 3 lcp-entry points: *get_block*, *release_block*, and *garbage_collect*, and a standard entry-point, *get_status*. The *get_status* operation is a non-blocking, read-only operation that exploits an s-thread's ability to read system-locked data without blocking. The operation reads the variables containing the allocation statistics which allows users to quickly obtain the approximate state of the allocator. The *get_block* operation allocates a backing storage block for a block in the object. The *release_block* operation releases all backing blocks beyond a specified block effectively truncating the file at that block. Both *get_block* and *release_block* update the allocation statistics to reflect the blocks allocated/released. Since *get_block* and *release_block* are lcp operations, and *get_status* is a read-only operation, the block allocator's internal data structures always appear to be in a consistent state.

A write call may append more than one block at a time, thus it is possible for a number of blocks to be allocated to a file object during a write operation, only to have that write operation abort. This would result in storage blocks allocated to an object but not used. The garbage collection entry-point can be called periodically to reclaim these blocks. The garbage collector requests from each object the list of blocks used by that object (using *get_block_ids*). Blocks allocated on the disk map but not used by any object can be reclaimed.

This implementation of file objects has atomic read/write semantics and trades off garbage collection for increased concurrency, fairness, and faster commit processing. Commit processing is faster since all commits affect only one object. Concurrency and fairness are increased since the block allocator object is never locked by any one thread for very long. The price we pay is that aborts become more expensive because they result in garbage that has to be collected. However, since commits should happen much more frequently than aborts, this should be a good trade-off. This example demonstrates how programmers using this paradigm can use a consistent set of object/thread semantics to make application-specific tradeoffs concerning concurrency, resilience, efficiency, fairness,

```
struct block {
        char data[BLOCKSIZE];
        int block_id;
}

class file {
        segment file_data {
                //# backed by allocator
                int numblocks;
                //real data;
                block filedata[ ]
        }

        segment block_data {
                //list of integers
        }
public:
        lcp-entry status read(int,
                char [BLOCKSIZE]);
        lcp-entry status write(int,
                char [BLOCKSIZE]);
        lcp-entry status truncate(int);
        lcp-entry status get_block_ids(id_list
                ids);
}
```

```
lcp-entry status
read(int position, int nblocks, char buf[nblocks*BLOCKSIZE])
{
        if (position + nblocks > numblocks)
                return(FAILURE);

        for (int i = position; i < position + nblocks; i++)
                strncpy(data[i], buf, BLOCKSIZE);

        return(SUCCESS);
}

lcp-entry status
write(int position, int nblocks, char buf[nblocks*BLOCKSIZE])
{
        for (int i = position; i < nblocks + position; i++) {
                if (i >= numblocks)
                        if (int block_id = get_block(i)) {
                                numblocks++;
                                // then add block_id to linked
                                // list of blocks_ids
                        } else abort();

                strncpy(data[i], buf, BLOCKSIZE);
        }

        return(SUCCESS);
}
```

Figure 4: Partial Definition of Robust File Object

1) determine which segment was being accessed
2) if the segment is not a data segment, jump to
   the real access violation handler
3) if access was a read
   a) check to see if a read lock is held on that segment
   b) if not, get read lock (call system-lock manager)
   c) set page table entries for the segment to allow reads
   d) mark read lock held on the segment
   e) return from interrupt
4) if write lock not held, get write-lock (call
   system-lock manager)
5) if thread is a gcp-thread of depth n and the version
   stack is of depth n, set the pte to allow write
   and return from interrupt
6) mark page in segment as shadowed, if not already marked
7) preserve current version by copying the page
8) push the version-stack record of the page
   (page #, thread depth, retrieval key)
   onto the version stack for that page
9) set protection mask on pte to allow write access
10) return from interrupt

Figure 5: An Access Violation Handler

and implementation complexity.

10 Implementation

The semantics presented here, while abstract, naturally lend themselves to being implemented in and supported by the operating system. The semantics are a super-set of the basic object/thread semantics as defined in section 2, and certain features such as the automatic system-level locking would be difficult and/or prohibitively expensive to implement at a language level.

This paradigm is being implemented as part of the Clouds v.2 operating system. User-level objects, object invocation, commit/abort processing, and system-level locking are being implemented as system-level objects in the Ra kernel. User objects are composed of Ra segments that contain either code or data. System-level locking is performed on *locking segments*. A locking segment may contain one or more (possibly discontinuous) Ra segments which may contain arbitrary variables or data structures. This allows flexible grouping of variables and data structures into logical locking groups.

The algorithms for handling nested and top-level cp-thread commits are adapted from nested and top-level action commit algorithms. However, since the system-level locking applies to persistent data residing in a single-level store (memory backed by secondary storage), system-level locking and version-stack management are being implemented using the virtual memory system and protection mechanisms. If a cp-thread does not hold a write (read) lock on a segment, the thread's page table entries (pte's) for that segment will be set to prohibit write (read or write) access. Attempts to access read/write or write-protected data results an access violation, whereupon control passes to the operating system access violation handler (see figure 5). The access violation handler then determines if the thread should be blocked or allowed to proceed, whether a shadow version must first be created, and which page table entries should be reset to allow read or write access. A simple access violation handler is shown in figure 5. This version shadows only the updated pages in a system-locking segment. A more sophisticated version would attempt to decrease the number of access violations by shadowing other pages around a touched page in anticipation of them being touched later.

11 Conclusions

This paper presents an object-based paradigm which supports a wide range of robust programming and has but one computation abstraction — the thread. Consistency labeling mechanisms and thread creation semantics may be used to achieve action/nested-action semantics. However, thread transformation can be used to achieve threads of execution that do not behave like actions but are globally consistent in certain segments, non-consistent (standard) in others, and locally consistent in others without having to supply complicated custom recovery schemes. This paradigm enables operating system designers to support robust and non-robust computation in a uniform manner while also giving object programmers fine control over the degree of consistency maintained within their objects and the methods used to achieve that consistency.

12 Acknowledgements

The example in section 9 resulted from a discussion with Rob McCurley. We also wish to thank Bill Appelbe, Glenn Benson, Rich LeBlanc, and the other members of the Distributed Systems Group at Georgia Tech for their feedback and suggestions.

References

[ABLN85] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[All83] J. E. Allchin. *An Architecture for Reliable Decentralized Systems.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983.

[AM83] J. E. Allchin and M. S. McKendry. Synchronization and Recovery of Actions. In *Proceedings of the Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ACM, Montreal, August 1983.

[B*85] K. P. Birman et al. An Overview of the ISIS Project. *Distributed Processing Technical Committee Newsletter, IEEE Computer Society*, 7(2), October 1985. (Special issue on Reliable Distributed Systems).

[BAHK*89] Jose M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahamad, William F. Appelbe, Partha Dasgupta, Richard J. LeBlanc, and Umakishore Ramachandran. The Architecture of *Ra*: A Kernel for *Clouds*. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, January 1989.

[BMS85] J. C. Berets, R. A. Mucci, and R. E. Schantz. Cronus: A Testbed for Developing Distributed Systems. In *IEEE Military Communications Conference*, IEEE Communications Society, October 1985.

[DLA88] P. Dasgupta, R. LeBlanc, and W. Appelbe. The Clouds Distributed Operating System. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[EGLT76] K. P. Eswaren, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in Database Systems. *Communications of the ACM*, 19(11):624–633, November 1976.

[Gra79] James N. Gray. Notes On Database Operating Systems. In *Operating Systems: An Advanced Course*, Springer-Verlag, Berlin, 1979.

[HMSC88] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–106, February 1988.

[HW87] M. P. Herlihy and J. M. Wing. Avalon: Language Support for Reliable Distributed Systems. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, July 1987.

[Jon79] A. K. Jones. The Object Model: A Conceptual Tool for Structuring Software. In *Operating Systems: An Advanced Course*, pages 7–16, Springer-Verlag, NY, 1979.

[LDH*87] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl. *Argus Reference Manual*. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1987.

[Lis82] B. Liskov. On Linguistic Support for Distributed Programs. *IEEE Transactions on Software Engineering*, SE-8(3):203–210, May 1982.

[LS79] B. W. Lampson and H. E. Sturgis. Crash Recovery in a Distributed Storage System. 1979. Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, CA.

[LS83] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[Mos81] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1981.

[PN85] C. Pu and J. D. Noe. *Nested Transactions for General Objects: The Eden Implementation.* Technical Report 85-12-03, Department of Computer Science, University of Washington, Seattle, WA, December 1985.

[SBD*84] A. Z. Spector, J. Butcher, D. S. Daniels, D. J. Duchamp, J. L. Eppinger, C. E. Fineman, A. Heddaya, and P. M. Schwarz. Support for Distributed Transactions in the TABS Prototype. In *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, pages 184–206, October 1984.

[WCC*74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, June 1974.

[Wil87] C. T. Wilkes. *Programming Methodologies for Resilience and Availability.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.

[WPLP85] Matthew J. Weinstein, Thomas W. Page, Jr., Brian K. Livezey, and Gerald J. Popek. Transactions and Synchronization in a Distributed Operating System. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 115–126, December 1985.