Implementing Consistency Control Mechanisms in the Clouds Distributed Operating System

Raymond C. Chen
Distributed Systems Group
Research and Development Division
Siemens-Nixdorf Information Systems
Cambridge, MA 02142

Partha Dasgupta
Department of Computer Science
and Engineering
Arizona State University
Tempe, AZ 85287-5406

Abstract
This paper presents an implementation of a kernel-level consistency control mechanism called Invocation-Based Consistency Control (IBCC). IBCC is designed to support general-purpose persistent object-based distributed computing. IBCC provides mechanisms that support a range of powerful, well-defined consistency semantics. For more sophisticated applications, the IBCC mechanisms can be used to implement custom recovery and synchronization.

The paper presents an operating-system level implementation of IBCC as part of the Clouds distributed operating system that uses memory faulting to initiate locking and intermediate version creation. The design takes into account the interactions that result from supporting distributed shared memory and can be implemented independently of the choice of the recovery technique. Performance aspects are discussed as well as the overhead incurred by supporting IBCC in terms of additional data structures needed in the operating system, and the additional amount of required code.

1 Introduction
This paper presents an implementation of a consistency control mechanism called Invocation-Based Consistency Control (IBCC) [7]. IBCC is designed for general-purpose, persistent object, distributed computing. IBCC provides mechanisms that can be used to achieve consistency ranging from no system-guaranteed consistency (process-based computing) to strict atomic transactions.

The IBCC mechanisms support powerful, well-defined consistency semantics. For more sophisticated applications, the mechanisms can be used to implement custom recovery and synchronization. Since consistency support incurs overhead, IBCC also supports low-overhead computation (the equivalent of processes), allows processes and transactions to co-exist and interact, and thus allows programmers to control the trade-offs between consistency and efficiency.

IBCC was designed to be implemented at the operating system level. This paper describes such an operating system-level implementation. The implementation uses the hardware virtual memory system to implement automatic read/write locking and version creation. The mechanisms have been implemented in the Clouds v.2 kernel (hereafter referred to as Clouds) as part of the Clouds distributed operating system.

This paper first presents an overview of Clouds and Invocation-Based Consistency Control. The paper then presents the design and implementation of the IBCC mechanisms and discusses the performance aspects of the implementation.

2 Related Work
A number of languages and transaction facilities have been proposed and some implemented that support general-purpose reliable distributed computing. Languages that support reliable persistent objects include Argus [16], Eden [1], Avalon [11], Arjuna [12], E [20], Trellis/Owl [18], and Napier88 [17]. Many of these language implementations use compiler-generated calls to a run-time support/storage system where the run-time system is either dedicated to supporting one language or some form of general-purpose stable storage manager or transaction facility such as Camelot [23], EXODUS [4], or Cricket [22]. Most of the above systems are implemented on top of existing operating systems such as Unix [21] or Mach [26]. Implementation of consistency mechanisms in a general-purpose operating system seems to be rare.

Locus [25] supports transactions over a reliable distributed file system. The Locus implementation uses logging on top of the shadowing-based reliable filesystem to provide fine granularity recovery. Clouds v.1 provided nested transactions, recoverable and non-recoverable data segments, custom locking facilities, and user-definable commit handling via exception handlers [9].

Our implementation relies heavily on using page fault/access violation exceptions to initiate lock requests and

---

*This work was supported in part by NSF grant CCR-86-19886.

1Unix is a trademark of AT&T
version creation. This memory reference exception technique has been used by others. The Bubba database system [3] and Cricket database storage system, and the 801 processor [5] use memory protections and access violations to set locks.

3 Clouds

The Clouds distributed operating system [8] is designed to support general-purpose distributed computing. Clouds provides data persistence and protection through the use of large-grained persistent objects. Computation is performed using threads (of control).

A large-grained persistent object is a long-lived address space that contains both code and data. Data stored in an instance of a large-grained persistent object is accessible only to the code within that instance. Threads execute concurrently within objects, and data stored in an instance of a object is accessible to all threads of control executing within that instance. The persistent object memory is therefore computationally shared. Clouds has no user-level concept of files or secondary storage. All persistent data resides in object memory which is automatically backed on secondary storage.

Threads enter objects by invoking an object-defined operation. Object invocation is location-transparent and may be nested and/or recursive. While a distributed system, Clouds has the look and feel of a centralized system.

Clouds is a native operating system written in C++ [24] and MC68020 assembler that runs on Sun-3 workstations. Clouds has been operational for over a year and is being used as an experimental testbed for research in distributed operating systems, the persistent object paradigm [10], distributed programming languages and environments [19], and reliable distributed computing.

4 Invocation-Based Consistency Control

Invocation-Based Consistency Control is a set of flexible mechanisms that control recovery and synchronization. Although strong consistency properties can be easily attained, the mechanisms themselves do not claim to guarantee consistency. Security can be traded off for flexibility. The intent is to use IBCC to build objects that appear to behave in a consistent fashion where "consistent" is application-dependent and to make recovery and synchronization features available to all users of the system, where a user is anyone or anything requesting operating system services. Users thus include compilers and run-time support systems for various languages. A detailed description of IBCC can be found in [7, 6]. We present an overview here for concreteness.

IBCC does not support the notion of a transaction or separate the notions of transactions and processes. Transactions and processes are unified into one computational entity, the thread, and the effects of both can be obtained through proper use of the mechanisms.

<table>
<thead>
<tr>
<th>Thread Label</th>
<th>Operation Consistency Label</th>
<th>GCP</th>
<th>LCP</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCP</td>
<td>No change</td>
<td>T → LCP</td>
<td>T → S</td>
<td>No change</td>
<td></td>
</tr>
<tr>
<td>LCP</td>
<td>T → GCP</td>
<td>T → LCP</td>
<td>T → S</td>
<td>T → LCP</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>T → GCP</td>
<td>T → LCP</td>
<td>No change</td>
<td>No change</td>
<td></td>
</tr>
</tbody>
</table>

Note: "T → " = "Thread transforms to"

† Though the label remains the same, this is a transforming invocation.

Figure 1: Consistency Label Compatibility

IBCC functions by providing a set of automatic recovery and synchronization mechanisms that control the visibility and permanence of altered data. These mechanisms are not explicitly invoked. Instead, the mechanisms come into play as a result of static labels placed on object methods by the programmer, grouping of data into locking units called locking segments, and extensions to the semantics of object invocation/return.

Threads carry a dynamic consistency label. Object operations are labeled with static consistency labels. When a thread invokes an object operation, the system checks the values of the thread and operation labels. Depending on the values of the labels, the thread label may change and commit processing may occur when the operation terminates.

4.1 Consistency Levels

The mechanisms of IBCC support three types of consistency as defined: global, local, and standard. Global consistency corresponds to multi-object consistency where a set of cooperating objects must be kept consistent both internally and with respect to each other. Local consistency corresponds to single-object consistency. Standard is the degree of consistency guarantee that programmers are currently used to — which is to say virtually none at all. With standard consistency, the system attempts to keep data consistent but makes no guarantees.

4.2 Operation Labelling

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

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

Each thread of execution in the system also bears a consistency label that is set to either GCP, LCP, or S. A thread is called an S-thread, GCP-thread, or LCP-thread if the thread's current consistency label is S, GCP, or LCP respectively.

When a thread with consistency label X invokes an operation T marked with consistency label Y, the invocation may be a transforming invocation depending on the value
of X and Y. Figure 1 summarizes the thread transformation scenarios and how the thread consistency label may change. An entry T \rightarrow A means that the label \( A \) is the result of the transforming invocation, and the value of the thread's consistency label changes to \( A \). If the entry is in bold, that indicates that the label will be 

A transforming invocation that causes the thread label to transform to some label \( L \) is called an \( L \)-transforming invocation or \( L \)-invocation for short.

The IBCC synchronization and recovery mechanisms behave differently depending on the value of the thread consistency label. With the exception of operations bearing the I label, the thread label always transforms to match the label on the operation.

A GCP, LCP, or S label on an object operation indicates a constraint placed on the execution of the operation by the programmer/compiler. The label indicates that the implementation assumes that the recovery/synchronization mechanisms will behave in a certain fashion.

The one exception to this is the I label. The inherited label indicates a lack of a consistency constraint. An inherited operation is supposed to be capable of supporting any type of consistency. Therefore, upon invoking an inherited operation, the thread label does not change. Instead, if the thread bears a thread consistency label \( L \), the operation behaves as if it had an \( L \) label. Thus an inherited operation invoked by a GCP-thread would perform automatic synchronization and recovery like a GCP-operation.

### 4.3 Version Model

IBCC synchronization and recovery lead to version stacks of persistent data where the older versions are at the bottom of the stack. The stack contains two types of versions: base versions and intermediate versions. The base version is the original version at the bottom of the version stack; it resides in secondary storage and is cached in primary storage (memory). Any intermediate versions (if they exist) reside in primary memory and may be backed by secondary storage. Primary memory is assumed to be volatile. Although secondary storage is assumed to be non-volatile, it is not assumed to be reliable. Each base version has an associated stable version that resides on stable storage. Thus, while base and intermediate versions may survive failures, only the stable version is guaranteed to survive a system failure.

### 4.4 Locking and Recovery

Threads executing GCP or LCP-invocations automatically lock any data (locking segments) they access before the access is allowed to proceed. This system-level locking is read/write and 2-phase. The operating system automatically requests a system-level lock on behalf of the executing thread/invocation when the thread attempts to read (or write) persistent data that it has not already read-locked (or write-locked). Locks gained during a transforming invocation are released only upon termination of the transforming invocation. Further details are available in [6].

Persistent data is grouped into user-defined locking granules called locking segments. The granularity of system-level locking is the locking segment. If a locking segment is locked, all data contained in that segment is locked. The system-level locking rules ensure that all version stacks created by GCP/LCP invocations are linear. Thus, there always exists a latest version.

S-threads always access the latest version of data. Unlike GCP/LCP invocations, S-invocations do not perform system-level locking. Therefore, S-threads never block as a result of IBCC synchronization. S-invocations can read uncommitted changes made by GCP/LCP-invocations. S-invocations can also overwrite data read/write-locked by GCP/LCP-invocations. If an S-thread makes a change to an intermediate version created by a GCP/LCP-invocation, that change gets committed/aborted when the intermediate version is committed or aborted as a result of the termination of the GCP/LCP-invocation. GCP/LCP-invocations may also read changes made by S-threads that exist in the base version but have not been committed to the stable version.

### 4.5 Commit Processing

GCP and LCP transforming invocations both perform commit processing when they terminate, although the scale of the commit processing is different. When GCP/LCP-invocations attempt to write to persistent data, the update is not immediately reflected in the stable version of the data. Instead, the system creates an intermediate version of the segment containing that data. Further updates to that data by the same transforming invocation are reflected in that intermediate version. When a GCP/LCP-invocation terminates (returns), all intermediate versions created by that invocation are committed or aborted automatically.

S-invocations perform no commit processing when they terminate. Like GCP/LCP-invocations, when an S-invocation attempts to write to persistent data, the update is not immediately reflected in the stable version, but rather in the latest version of the segment. When a GCP/LCP-invocation terminates (returns), all intermediate versions created by that invocation are committed or aborted automatically.

GCP and LCP transforming invocations can be thought of as creating recovery blocks. A GCP-invocation creates a dynamic recovery block that may include multiple objects. An LCP-invocation creates a static recovery block that includes only one object.

### 5 Implementation Assumptions

This implementation design makes two assumptions about the underlying hardware. First, the address space of the machine can be broken up into a number of address ranges that can be individually read/write-protected. Second, when an attempt is made to access a protected address range in a manner prohibited by the protections, the operating system gains control via an exception handler and the hardware indicates whether the attempted access was a read or write.
These assumptions are necessary as the design calls for using the protectable address ranges to implement locking segments and for using memory exceptions to initiate system-level locking and version creation. Each locking segment is implemented using one or more protected address range. The implementation architecture presented in this paper is designed to be implemented on paged virtual memory architecture, or segmented/paged virtual memory architecture like the Sun-3.

6 Virtual Memory and the Single-level Store

In file-based environments, persistent data is accessed through reads and writes of files/relations, and the operating system or database gains control at each read and write request. This allows the system to intercept reads and writes of persistent data, and any necessary locking or recovery processing is executed.

However, in a system using a single-level store such as Clouds, persistent data is accessed through direct reads and writes of primary memory, not through operating system requests. Thus, the operating system can not intercept reads/writes of persistent data unless a memory fault (page fault or access violation), software exception (divide by zero, etc.) or hardware interrupt occurs.

The implementation therefore uses the virtual memory system protection mechanisms to initiate locking and version creation. Page-level read/write protection is used to intercept reads and writes of unlocked data. If a GCP/LCP invocation does not hold a read (or write) lock on a locking segment, the pages of that locking are set to disallow reads (or any access). Attempting a disallowed access results in an access violation allowing the operating system to take control. Read-faults when accessing a segment are interpreted as requests for a read-lock on the segment, and write-faults are interpreted as requests for a write-lock on the segment. Writing to a write-locked page results in an intermediate version of that page being created.

7 Functional Behavior

The IBCC implementation performs the following tasks. When terminating transforming invocations, performs recovery processing when terminating transforming invocations, implements the implicit system-level locking, automatically creates intermediate versions when necessary, allows S-invocations to access the latest versions of all data without blocking. These functions are initiated by object invocation, page faults, or object return. This section describes the major data structures in the IBCC implementation and the flow of control of object invocation, page faulting, and object return.

7.1 Major Data Structures

The implementation must maintain a number of important data structures. The three major data structures are the object header (also known as a Virtual Space Descriptor), the segment table, and the Transformation Segment. Each Clouds object has an object header that is used by the kernel when handling page faults. Clouds objects are composed of segments and the object header defines size and location of each segment, the protection mask on each segment (no access/read-only/full access), and the consistency label for each object entry-point. For the purposes of IBCC system-level locking, each segment is a locking segment.

Each Clouds segment has a segment table with it that indicates which segment pages/blocks are held in which physical page frames. The page frames referenced by the segment table always contain the latest version of the segment.

Each thread has a transformation segment (t-segment) associated with it. A t-segment contains information on the locks held, objects visited, and intermediate versions created by every non-terminated transforming invocation initiated by the thread.

7.2 Object Invocation

On each local object invocation, the kernel checks the object header to get the consistency label of the entry-point being invoked. The consistency label is then checked against the thread's current consistency label to determine if the invocation will be a transforming invocation. If so, the transforming invocation is recorded on the t-segment associated with the thread.

If the invocation is an S transforming invocation (from GCP/LCP to S), no further IBCC-related processing occurs, and the local invocation proceeds as usual. If the invocation is a GCP/LCP transforming invocation, a shadow object header is created from the header of the object that is being invoked. The shadow object header is identical to the object header except that the memory protection masks on all persistent data segments are set to disallow any access to the data. The object invocation then proceeds using the shadow object header to control the memory mappings in place of the actual object header.

7.3 Access Violations

If a read/write of a segment causes an access violation, the kernel determines if the invocation holds a system-lock on the segment. If no lock is held, the kernel obtains a lock on behalf of the invocation. If the access is a write, the kernel must also create a new version of the segment, which entails preserving the current version (discussed in further detail in section 8.2). The kernel then sets the memory mappings in the (shadow) object header to allow the attempted access.

7.4 Object Return

On an object return, if the terminating invocation is a transforming invocation, the thread consistency label (as well as
other attributes, see [8] for details) is reset. If the terminating
invocation is a GCP or LCP invocation, the kernel then
performs any necessary commit processing.

8 DSM and IBCC

The Clouds system supports distributed shared memory
(DSM) [15] to allow threads on several sites to simultane-
ously access the same objects. Although multiple threads on
different sites may be reading or writing the same page, a
DSM coherence protocol similar to a multi-processor cache
coherence protocol ensures that single-copy semantics are
preserved. The IBCC implementation must function in a
DSM environment. Integrating IBCC and DSM necessitated
adding functionality to DSM, and requires the use of dis-
tributed locking.

8.1 Distributed Locking and Shared
Memory

In a persistent object system supporting distributed shared
memory, local locking is insufficient. In system with DSM,
two GCP-threads on different sites may attempt to update
the same locking segment. When this happens, only one
thread can gain the lock; the other thread must block. Lock-
ing requests and locking information must therefore be man-
aged globally.

Distributed locking can be implemented in two ways:
by adding locking to the DSM system or by using a Dis-
tributed Lock Manager. This implementation uses a Dis-
tributed Lock Manager. The DSM system is not responsible
for locking, only for maintaining coherence.

Using a separate Distributed Lock Manager has three
major advantages. A separate, orthogonal Distributed Lock
Manager can support both the system-level locking required
for invocation-based consistency control and a standard user-
accessible read/write locking service. Furthermore, by iso-
lating the two functions (DSM coherence and distributed
locking), either system can be experimented with or altered
without affecting the other. This is a significant advantage
as Clouds is being used as a research testbed.

In this design, the logic that determines whether system-
level locking must be performed can be isolated in a third
system, the Virtual Memory Manager. Neither the Dis-
tributed Lock Manager nor the DSM coherence system
need know about consistency labelling or the S/LCP/GCP
semantics. If distributed locking were implemented in the
DSM coherence system, the DSM coherence system
would have to be augmented to implement not only IBCC
read/write locking but also the semantics of interactions be-
tween S-invocations and GCP/LCP-invocations.

8.2 Version Management and DSM

As discussed earlier, IBCC presents a version stack-based
model of recovery to the user. To correctly implement IBCC,
the system must properly create, commit, and abort inter-
mediate versions. Once an intermediate version is created,
the system must also preserve the contents of older versions
until they are overwritten as a result of a commit.

Having DSM in the system adds further complexity. In a
system without DSM, the operating system on a node knows
whether an object is stored locally or remotely. In a system
with DSM, the DSM coherence system hides this knowledge
from the rest of the operating system. An object controlled
by DSM may reside anywhere in the distributed system but
appears to all sites to be stored locally. If a computation
executing on the local site alters a set of pages controlled
by DSM (thus moving them to the local site), the pages
may have been moved to one or more other sites and not be
present locally at the time commit processing begins. This
design therefore calls for augmenting DSM to handle ver-
sion creation, precommit, commit, and abort processing for
DSM-controlled segments.

Local recovery processing is controlled by a Recovery
Management System. Page fetch and recovery requests are
handled by a Recovery Control System which makes requests
on the local Recovery Management System or on the DSM
system or both, depending on the nature of the request.

When directed to create a version, the DSM system must
create a new version of the segment and preserve the state of
the current segment version. Due to the fact that S-threads
can access data without blocking, to preserve single-copy
semantics, every page in the current version of the segment
must be preserved before allowing the GCP/LCP invocation
that created the new version to proceed. Likewise, commits,
precommits, and aborts affect every page in the segment
version. These operations on a segment therefore involve
processing on every site that holds a copy of a page of that
segment.

9 Implementation Architecture

An implementation of invocation-based consistency control
must correctly perform the following:

- On each object invocation, determine if the invocation
  is a transforming invocation.
- Intercept reads/writes of unlocked persistent data by
  GCP/LCP-invocations, interpret the accesses as lock
  requests, and process the lock requests.
- Properly create, commit, and destroy intermediate
  versions.
- Ensure that S-invocations always access the latest ver-
  sion of persistent data without blocking.

The IBCC implementation architecture (shown in figure
2) consists of an Invocation Processing Layer, Object
Information System, Transformation Information System,
Page Information System, Distributed Lock Manager, Vir-
tual Memory Manager, Object Invocation System, and a Ver-
sion Manager consisting of a Recovery Control System, DSM
System, and Recovery Management System.

The Invocation Processing Layer and Virtual Memory
Manager are the control layers. They gain control on object
invocation/return and page faults. The remainder of the
components can be classified as support systems that provide services required by the control layers. The remainder of this paper describes the functional descriptions of each of the sub-systems, control layers, and their implementation in the Clouds kernel.

9.1 Version Manager

The Version Manager is responsible for moving versions (of pages) between primary memory and secondary storage, and for correctly maintaining stable storage. This includes creating, committing and aborting intermediate versions, flushing base versions to stable storage, and fetching the latest version from secondary storage. In this architecture, the Version Manager consists of a Recovery Control System (RCS), DSM system, and Recovery Management System (RMS).

The Version Manager provides a get() routine that returns a page frame containing the contents of the latest version of that page. The Version Manager ensures that all fetches of the latest version (get()) get the new version.

The Version Manager also supports create(), commit(), and abort() for managing intermediate versions. The create() function informs the Version Manager that a new intermediate version of a segment is being created and that the state of the current version should be preserved.

The commit() routine commits the latest version. The Version Manager then ensures that all threads using the latest version access the newly committed version. Similarly, abort() deletes the latest intermediate version. All threads working on the latest version will then automatically access the next-oldest version.

9.1.1 Version Manager Implementation

In the current configuration of the Clouds system, all persistent objects are backed by data servers. The data servers use the Unix filesystem to store Clouds object images. A similar scheme is used to implement paging/swapping devices for Clouds sites.

The current implementation commits pages by writing the versions back across the network to the appropriate data server. If multiple data servers are involved, the pages are committed sequentially, one server at a time. This scheme leaves two points of vulnerability. First, if the Unix filesystem fails, the data will be lost. Second, if a commit involves multiple data servers, a failure during commit processing...
may result in a partial commit. Well-known techniques exist for addressing the commit and reliable storage problems [13, 14]. However, given that Clouds is being used for research as an experimental testbed, it was felt that implementing these techniques would not have been cost-effective.

9.1.2 Other Systems

We now present a brief overview of some of the remaining systems in the implementation.

- The Virtual Memory Manager (VMM) implements automatic system-level locking and initiates intermediate version creation. (Shadow version destruction via commit/abort processing is handled by the object return mechanism.) In the Clouds implementation, the Virtual Memory Manager is implemented directly in the page fault handler. The Clouds Virtual Memory Manager is approximately 700 lines of code. Adding the logic necessary to support IBCC required adding approximately 100 lines of code to the Virtual Memory Manager.

- The Invocation Processing Layer (IPL) controls the initiating and terminating of transforming invocations. The IPL gains control at the commencement and termination of every object invocation. The IPL must handle local and remote object invocation and object return. The Invocation Processing Layer as originally implemented consisted of approximately 1000 lines of code. This code included the remote object invocation facility. Supporting the functionality required by invocation-based consistency control required an additional 300 lines of code as well as an extra 24 bytes of data in the remote invocation control message.

- The Object Invocation System is responsible for properly saving and restoring the virtual memory mappings and hardware state such as registers necessary for an object invocation, as well as maintaining a per-invocation attributes. These attributes include the consistency label of the invocation whether the current invocation is a transforming invocation. The original implementation of threads and object invocation consisted of approximately 3000 lines of code. The modifications required entailed adding less than 50 lines of code and two integers to the invocation record that is pushed and popped on object invocation/return.

- The Object Information System maintains the object headers and shadow object headers, and makes the information available in the headers available to the rest of the kernel. The kernel code necessary to implement and manage object headers total approximately 2500 lines of code. Modifying the original design to support the additional required functionality involved adding less than 100 lines of code and 160 bytes to the object header. The current object header allows for an object to define up to 140 entry points. Each entry point requires one extra byte in the object header.

- The Transformation Information System (TIS) maintains t-segments. Although each thread has an associated transformation segment, the segment is not accessed unless the thread invokes a transforming invocation. The TIS creates t-segments only when necessary, therefore threads which perform only S-invocations incur no overhead as a result of t-segment management.

The Transformation Information System had to be added to the system and required approximately 1000 lines of code to implement.

- The Page Information System (PIS) maintains information on whether a page of a segment/object is present in the system, whether the page is present in physical memory, and the page frame containing the page if present in physical memory. The PIS is updated when pages are invalidated or moved between memory and secondary storage (either local or remote disk).

The Page Information System was already necessary for the proper functioning of the Clouds system and required 900 lines of code to implement.

- The Distributed Lock Manager provides distributed read/write locking. Locking is performed on locking names, (name, integer) and are granted on the basis of locking ids, (thread name, integer). This enables the lock manager to provide conventional read/write locking for threads as well as and the system-level locking required by invocation-based consistency control.

The Distributed Lock Manager required approximately 3500 lines to implement of which approximately 2500 lines are shared with an implementation of distributed semaphores.

10 Performance and Overhead

In keeping with the IBCC philosophy of supporting low-overhead computing, one goal of the implementation design was to minimize the amount of overhead incurred by S-invocations in an IBCC system as compared to the overhead incurred by threads in a similar non-IBCC system. The majority of the performance overhead is incurred as a result of the distributed locking, version creation, and commit/abort processing performed by GCP/LCP-invocations. However, this overhead is necessary for correctness and is incurred only by the GCP/LCP computations desiring the consistency provided by Invocation-Based Consistency Control.

The measurements given in this section were taken on a Clouds kernel running on a Sun-3/50 with a 12 MHz 68020 processor using a microsecond timer chip. The Unix utilities (DSM server and lock manager) were running on a Sun
Measurements indicate commit times of approximately 33 msec per 8K page. This time reflects the time necessary to reliably send the page to a Unix server process acting as a network disk and wait for the acknowledgement.

Measurements indicate that lock requests on the Distributed Lock Manager which must be forwarded to a remote site require approximately 9 msec to complete, and lock requests that can be serviced locally require under a network disk and wait for the acknowledgement. Approximately 1.3 msec of this time is spent getting into and out of the kernel (via the page fault handler) and determining the segment to be locked.

Committing 3-page segment requires 131 msec. The time breaks down into 33 msec per altered page (to send the page to the DSM owner site), plus 9 msec to unlock the segment (assuming the lock is managed off-site), plus 11 msec to send and process the control message to the DSM owner site that signals the segment commit, plus 12 msec of miscellaneous processing.

Creating a version (and preserving the current version) of a segment page requires approximately 33 msec. As in the commit processing, the majority of this time is spent communicating across the network. In this case, an 8K page is transmitted across the network. Either the faulting site sends the current version of the page to the DSM owner site to be preserved or the DSM owner site sends the latest version of the page to the faulting site.

Querying the Transformation Information System to determine if a lock is already held by an invocation requires 0.1 msec. If a lock must be gained during a fault in addition to creating a new version, servicing the fault requires an additional 2 msec or 9 msec depending on whether the lock is controlled locally or remotely.

Neither the Clouds kernel nor the Unix servers were optimized or tuned. Recent measurements on a tuned version of the kernel yielded 16 msec for per 8K page transmission and 4 msec to transmit a control message. IBCC performance times should thus drop drastically once the performance improvements that yielded these measurements are merged into the main Clouds release. Commit times per page should drop to between 16 and 19 msec. Remote lock request times should drop to approximately 5 msec.

11 Acknowledgements

The Distributed Lock Manager discussed in this paper was implemented by R. Ananthanarayanan. The remote object invocation facility and a portion of the Object Invocation System were originally implemented by Mark Pearson. We would also like to thank Sathis Menon, Ajay Mohindra, Chris Wilkenloh, for their efforts in adding to the functionality of the Clouds kernel.\(^3\)

References


---

3Bibliography truncated due to length limitations. References available on request.