A Support Architecture for
Reliable Distributed Computing Systems

Final Project Report
May 15th 1990

From:
Georgia Tech Research Corporation
Atlanta, Georgia 30332

To:
National Aeronautics and Space Administration
Langley Research Center

Grant No.: NAG-1-430

Endorsements:

Partha Dasgupta
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2572

Richard J. LeBlanc Jr.
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2592
A Support Architecture for Reliable Distributed Systems

1. Introduction
The Clouds project at Georgia Tech has been funded by NASA since 1981. During this period we have designed and implemented a distributed object-based operating system, which has a novel structure and supports an unconventional systems-programming style.

The NASA funds received this far had a major impact on the formation of the Clouds project and in the area of research in distributed operating systems. Using these funds we have achieved recognition in the research community, published several papers and many technical reports, implemented the first version of the Clouds system, and most notably, were selected by the National Science Foundation as one of the recipients of the 1987 CER (Coordinated Experimental Research) grants.

The first Clouds kernels was a monolithic, native kernel that ran on VAX-750 computers. This version was implemented in the 1983-86 timeframe and was used till mid 1987. After this kernel served its purpose, we redesigned the implementation and have built a new minimal kernel. The redesign effort was started in late 1987. Since 1988 we have redesigned Clouds, implemented the system as a minimal kernel, added system services and user service and have built a usable distributed system. The current version runs on Sun 3/60 machines.

This report provides a brief overview of the Clouds implementation and the project status.

2. The Clouds Project
The Clouds project is a distributed operating systems project. The goals of the project are:

- The operating system will be distributed over several sites. The sites will have a fair degree of autonomy. Yet the distributed system should work as an integrated system. Thus the system should support location independency for data, users and processes.
- To keep the system simple, the user environment must be built around a straightforward, simple and elegant paradigm. The paradigm must encapsulate storage, processing and distribution.
- The users should be shielded from both the configuration of the system (site independence) as well as its failure modes. For example, if the site a user is connected to fails, he should be transferred to an active site transparently.
- Reliability is a key requirement. Large distributed systems use significant number of hardware components and communication interfaces, all of which are prone to failures. The system should be able to function normally even with several failed components. Data should be kept consistent at all times.
Distributed systems often have dynamic configurations. That is, newer hardware gets added, or faulty hardware is removed. The system function should not be hampered by such maintenance chores. Thus the system should be dynamically reconfigurable.

Many of the above functions can be implemented on conventional systems, but would make the system extremely slow. Thus efficiency is an important design criteria.

The above goals have resulted in a elegant and feasible design of a object-based distributed operating system.

3. The Design of Clouds
The Clouds design hinges on a simple paradigm of system structure, using two basic primitives namely objects and threads. The following are the salient points of the design.

- An object-based, passive system paradigm is used as the basic architecture. All system functions, data, user programs and resources are encapsulated in passive objects. These objects can be invoked at appropriate entry points by threads.

- Threads and the only active entities in the system. A thread is a path of execution that visits entry points in objects in a procedure-call fashion and may span several objects in several sites. Also several threads may concurrently execute in the same object.

- The objects in Clouds represent nearly everything the system has to offer. The site independence philosophy is implemented by making the object name space (system names) flat and site independent. When a thread on any machine invokes an object located anywhere, no site names are used. Hence the location of any particular object is unknown to a thread.

- Efficiency has been of concern. To this end the Clouds kernel (also called Ra) is a small minimal kernel that support threads and object invocation. The objects are handled by the virtual memory system. This design is kepts simple without losing generality of flexibility, and is expected to be efficient.

The Clouds paradigm has been augmented to support features for reliable computation. The two major features added to support reliability is the notion of consistency-preserving invocations and replicated computations.

- The consistency support system is a user programmable set of tools which are triggered by declaring the entry points of the objects with a consistency level. The consistency levels include global, local and standard. These labels give rise to threads (upon invocation) that have specific semantics. A complete outline of the semantics is outside the scope of this report. The allowed semantics provide the support needed for writing robust programs.

- Reliability is further enhanced with replication at the data as well as computation level. This allows the system to recover from a certain fixed number of faults with minimal overhead.
4. The implementation of Clouds

The implementation of Clouds is largely complete. The implementation includes a minimal kernel called Ra. The Ra kernel provides low level memory management and scheduling support. The rest of the operating system is programmed at above the kernel level using a facility called system objects. The following is a brief outline of the services provided by the system objects:

1. **Buffer Manager**: This is a low-level service that handles the buffer allocation, deallocation, packaging and unpackaging needs of a variety of system service routines.

2. **Object Manager**: The Ra kernel provides the mechanisms necessary for handling objects in the operating system. The Object manager handles most of the policy including construction, activation and invocation of objects.

3. **Network Driver**: The basic service used by all distributed facilities of the operating system is the Ethernet driver that allows packet communication over a local area network.

4. **The Transport Protocol (RaTP)**: RaTP is a reliable message transport protocol that is used by all communication activities. RaTP is a connectionless, lightweight protocol that can handle message transaction oriented (synchronous) communication. This is used to support a variety of functions detailed below.

5. **Thread Manager**: A Clouds thread is a distributed computation. The thread manages handles the remote processing abilities of threads.

6. **RPC Handler**: The Clouds remote procedure call facility is built on top of RaTP in conjunction with the thread manager. Allows remote objects to be executed.

7. **Network Disk Manager**: The Clouds machines are equipped to run diskless. The disk services are provided from file-servers. Thus the network disk drivers create virtual disks for object storage and swapping for Clouds machines. The Network disk drivers run on top of RaTP.

8. **Swap Manager**: The local swapping facilities of each Clouds machine is handled through the swap manager. The swap manager implements the paging facilities and ties them to a swap partition on a remote disk via the network disk manager.

9. **DSM Partition**: Clouds supports a Distributed Shared Memory facility. This makes all objects in the system appear local to all machines running Clouds. The DSM partition provides this virtual object access capability along with coherence support to enforce single-copy semantics.

10. **Distributed Locking Handlers**: Objects have local control over synchronization of access to its private data, even when accessed by several threads running on different machines. This is handled by the distributed locking service. The distributed locking service is accessed by programmers as semaphores declared within the object.

11. **User I/O Handler**: User I/O is handled by Clouds objects as ASCII I/O to user terminals. User terminals are in fact windows running under SunWindows on Unix machines. The user can run Clouds applications via a Unix window. All I/O
is redirected to the appropriate window over the network. The User I/O manager supports printf’s and scanf’s from objects to the user pseudo-tty.

In addition to the above services, Clouds provides a modified C++ programming environment called CC++. We support a CC++ compiler and mechanisms of generating objects and instances.

5. Progress Report

The Clouds research team currently consists of five academic faculty, two research scientists and about 8 graduate students. Most of the funding is from the National Science foundation. The NASA support is used to augment the NSF support and currently it supports one Ph.D. student and some faculty time.

The above system functions have been implemented and tested. We are currently augmenting the implemented functions with support for programming environments and consistency mechanisms. Design work in the areas of naming, location, object-oriented programming support and distributed management is in progress.

6. Bibliography

The following publications include contributions by persons supported by NASA:


Appendix

Selected Papers
A novel system architecture, based on the object model, is the central structuring concept used in the Clouds distributed operating system. This architecture makes Clouds attractive over a wide class of machines and environments. Clouds is a native operating system, designed and implemented at Georgia Tech, and runs on a set of general purpose computers connected via a local area network.

The system architecture of Clouds is composed of a systemwide global set of persistent (long-lived) virtual address spaces, called objects that contain persistent data and code. The object concept is implemented at the operating system level, thus presenting a single level storage view to the user. Lightweight threads carry computational activity through the code stored in the objects.

The persistent objects and threads gives rise to a programming environment composed of shared permanent memory, dispensing with the need for hardware-derived concepts such as file systems and message systems. Though the hardware may be distributed, and may have disks and networks, the Clouds provides the applications with a logically centralized system, based on a shared, structured, single level store.

The current design of Clouds uses a minimalist philosophy with respect to both the kernel and the operating system. That is, the kernel and the operating system support a bare minimum of functionality. Clouds also adheres to the concept of separation of policy and mechanism. Most low-level operating system services are implemented above the kernel and most high level services are implemented at the user level. From the measured performance of using the kernel mechanisms, we are able to demonstrate that efficient implementations are feasible for the object model on commercially available hardware.

Clouds provides a rich environment for conducting research in distributed systems. Some of the topics addressed in this paper include distributed programming environments, consistency of persistent data and fault-tolerance.

1. Introduction

The Clouds project is an ongoing distributed operating system project at Georgia Tech. The Clouds operating system was first designed in 1983 [Al83, Mc84]. The implementation was started in 1984 and was completed in 1986 producing the first version of the software (Clouds v.1) [Sp86, Pi86, Ke88]. In mid-1987 we started designing the second version of Clouds and the kernel for this version has been completed in mid-1988 [DaLeAp88, BeHuKh88, BeHuKh89]. This version is called Clouds v.2. In this paper we discuss the basic philosophies and techniques that have been developed through the experience gained in the Clouds project as well as the implementation and research issues.

* This research was partially supported by NASA under contract number NAG-1-430 and by NSF grants DCS-8316590 and CCR-8619886 (CER/II program).
What can objects do?

Objects can be used for most everything. An object is an encapsulation that includes not only long-lived data; but a set of operations (invocations) on the data, or an environment under control of the object.

In the simplest form, objects can be considered modules that provide a function or service. For example a shared library, a data server, a program fragment and so on.

Objects can also provide specialized services. For example, a sensing device can be represented as a object and an invocation can be used to gather data from the device, without having to know about the mechanisms involved in accessing the device, or even the location of the device. Similarly, terminal I/O is effectively handled by a terminal object (with read and write operations defined on it).

Objects can be active. If a process is started in a infinite loop in an object, this process can monitor the environment of the object and inform some other entity (another object) in case of need. This is particularly useful in conjunction with an object managing sensory monitoring devices.

Objects are a simple concept with a major impact. From general purpose programming needs to quite specialized applications, the object can be used for almost every need, and yet provide the same, simple procedural interface to the rest of the system.

The paradigm used for defining and implementing the system structure of the Clouds system is an object/thread model (section 2). This model provides threads to support computation and objects to support an abstraction of storage. This model has been augmented to support atomicity of computation and to provide support for reliable programs [Al83, ChDa89, AhDaLe88].

1.3. Project Overview

The first version of the Clouds operating system has been implemented and is operational. This version is referred to Clouds v.1. This was used as an experimental testbed by the implementors. This implementation was successful in depicting the feasibility of a native object-based operating system, supporting the Clouds paradigm. The implementation of Clouds v.1., in retrospect had some flaws that resulted in poor performance and non-portability of the kernel. Experience with Clouds v.1. taught us that the approach works; it also taught us how to do it better.

The lessons learned from the previous implementation have been used to redesign the kernel and build a new version called Clouds v.2. The basic system structuring paradigm used in v.1. and v.2. remain the same. Most of the design objectives mentioned earlier are the same as before. However some of the goals and most of the design and implementation of the system has changed. Clouds v.1 was targeted to be a testbed for distributed operating system research. Clouds v.2. is targeted to be a distributed computing platform for research in a wide variety of areas in Computer Science system research.

The structure of Clouds v.2 consists of a minimal kernel called "Ra", and a set of system-level objects providing the operating system services. Ra supports a set of basic function of the system: virtual memory management, system object support and low-level scheduling. The system objects provide other systems services (user object management, synchronization, naming, atomicity

* After the Egyptian Sun-God.
Page missing from report
segment is accessible by the code in the code segment, but not by any other object. Thus the object has a wall around it which has some well-defined gateways through which activity can come in. Data cannot be transmitted in or out of the object freely, but can be moved as parameters to the code segment entry points (see the discussion on threads). A Clouds object is shown in Figure 1.

Clouds objects can be defined by the user or defined by the system. Most objects are user-defined. Some examples of system-defined objects are device drivers, name-service handlers, communication systems, systems software, utilities, and so on. The basic kernel is not an object; it is an entity that provides the mechanism for the support of object invocations. A complete Clouds object can contain user-defined code and data, system-defined code and data that handle synchronization and recovery, a volatile heap for temporary memory allocation, a permanent heap for allocating memory that will remain permanent as a part of the data structures in the object, locks, and capabilities to other objects.

Though Clouds objects can be created, deleted and manipulated individually, the operating system is designed to support a class and instantiation mechanism. This mechanism allows users to derive object-classes from a set of parent classes, as well as create object-instances of an object-class. Further details are dependent on the programming language and environment used, and are beyond the scope of this paper.

2.2. Threads

The only form of activity in the Clouds system is the thread. A thread can be viewed as a thread of control that executes code in objects, traversing objects as it executes. Threads can span objects, and can span machine boundaries. In fact, machine boundaries are invisible to the thread (and hence to the user). Threads are implemented in the Clouds system as lightweight processes that have a stack space but no data space. A thread that spans machine boundaries is implemented by several processes, one per site.

Upon creation, a thread starts up at an entry point of an object. As the thread executes, it executes code inside an object and manipulates the data inside this object. The code in the object can contain a call to an operation of another object. When a thread executes this call, it temporarily leaves the calling object, enters the called object, and commences execution there. The thread returns to the calling object after the execution in the called object terminates. The calls to object entry points are called object invocations. Object invocations can be nested. The code that is accessible through an entry point is known as an operation of the object.

A thread executes by invoking operations defined inside many objects. Unlike processes in conventional operating systems, a thread often cross boundaries of virtual address spaces. Visibility within an address space is, however, limited to that address space, thus the thread cannot address any data outside its current address space. Control transfer between address spaces occurs through object invocation and data transfer between address spaces occurs through parameters to object invocation. A thread in
Page missing from report
Second, the storage mechanism used in this object-based environment is quite different from that used in the conventional operating systems. Conventionally, the file is the storage medium of choice for data that has to persist, especially since memory is tied to processes and processes can die and lose all the contents of their memory. However, memory is easier to manage, more suited for structuring data and essential for processing. The object concept merges these two views of storage, to create the concept of a set of shared, structured, permanent virtual spaces.

Just as Clouds does not have files, it does not provide user-level support for disk I/O. In fact there is no concept of "disks" or such I/O devices (except user terminals). The system creates the illusion of a large virtual memory space that is permanent (non-volatile), and thus the need for using peripheral storage from a programmer's point of view, is eliminated.

Many distributed systems are message-based, and hence use messages as the paradigm of choice. In the object-thread paradigm, like the need for I/O, the need for messages is eliminated. Threads need not communicate through messages. Messages can be simulated using objects implementing bounded buffers for applications written using a message-oriented algorithm. Thus ports are not supported. This allows a simplified system management strategy as the system does not have to maintain linkage information between threads and ports.

Objects provide an easy to use abstraction for shared memory, a special case of which ("problem-oriented shared memory") is recommended by Cheriton as a powerful tool for programming distributed systems [Ch86]. Shared memory (consistent or not) is seen by many as a better concept than messages for programming distributed systems, e.g. Linda [Ge85].

To summarize:

- The Clouds system is composed of named address spaces (objects).
- Activity is provided by threads moving amongst the population of objects through invocation.
- Data flow is implemented by parameter passing.

The system thus looks like a set of permanent address spaces which support control flow through them, constituting what we term object memory.

The basic paradigm discussed above is the common link between Clouds v.1. and Clouds v.2. Both are identical in this regard. Most of the other features of the two are different.

3. Clouds v.2.

The development of the second version of Clouds was undertaken for several reasons:

- The implementation of Clouds on the VAX machines suffers from inherent problems due to the VAX architecture. Other machines are better suited for implementing an object system due their virtual memory mechanisms being more compatible with the demands of a persistent object system [RaKh88b].
- The Clouds v.1. kernel was not written to be portable. Since the VAX did not prove to be an effective target machine for Clouds, porting entailed major rework.
- The kernel was monolithic, large and hard to modify. This attribute encouraged us to think about the minimal kernel approach.
- We learned how to manage virtual memory better, and could identify a lot of situations where the need for separation of policy and mechanism was necessary for further research into both policy and mechanisms.

The redesign of Clouds not only prompted a redesign of the kernel, but a rethinking of all of the system functions. The only aspect of the system that remained unchanged in the redesign phase was the basic object/thread paradigm described in sections 2.
Page missing from report
tual machine is shown in Figure 3. The abstractions supported by the Ra virtual machine are:

- **Segments:**
  A segment is a contiguous block of virtual memory and its contents. The contents of a segment are an uninterpreted byte-sequence. Segments are explicitly created and persist until destroyed. They have systemwide unique sysnames and they have storage attributes, such as zero-filled, volatile, persistent, recoverable etc.

- **Windows:**
  A window defines a contiguous range of addresses in a virtual space. This range is associated with a range of contiguous addresses in a segment. Windows are used to create virtual spaces out of segments. Windows have protection attributes such as read-only, read-write and so on.

- **Virtual Spaces:**
  A virtual space is the abstraction of a complete address space. A virtual space consists of a set of windows into one or more segments. Thus a virtual space is a set of subranges of segments, coalesced into one address space. The virtual space is the representation of an object, when the object is being actively used. The composition of a virtual space is shown in Figure 4.

- **Partitions:**
  Partitions are containers for segments and provide non-volatile data storage for the segments. A segment lives in exactly one partition. When a segment is in use, it can be demand paged, in and out, of its partition. Partitions can exist on any machine in the network, including machines not running Ra. (e.g. Ra segments could reside on a Unix machine running a Ra partition server.) Though partitions are a part of the Ra virtual machine, partitions are imple-
Page missing from report
The structure of the system memory that contains the kernel, system objects and kernel classes is shown in Figure 5; Figure 6 shows the storage and activity hierarchies in Clouds; and the structure of the Clouds/Ra system is shown in Figure 7. The design of Ra is explained in further detail in [BeHuKh89].

### 3.1.3. Objects in v.2.

The Clouds objects are implemented in v.2. through a system object called the object manager. The object manager manages objects, creates and deletes them and provides the object invocation facility.

An object is a set of windows into a set of segments and is implemented using a Ra virtual space. The information regarding the windows and segments is stored in another segment called the object descriptor. The sysname of the object descriptor segment, plus requisite access rights is the capability of the object.

Objects that are in Ra virtual spaces can be invoked. Information on the most recently used objects is contained in an activated object table in the object manager. When a thread invokes an object, the object manager first checks the object table. If the information is present the invocation proceeds. Otherwise the object table information is built by reading in the object descriptor.

### 3.1.4. Invocation using RPC and DSM

The object naming scheme in v.2. is the same as in v.1. (system-wide capabilities). The remote object location scheme is currently a subject of research. At present we are using the cache/search/invoke scheme used in v.1. but may change to a multicast-based scheme [Pi86, Sp86, AhAm87, Be88].

The object invocation scheme is handled by the object management system (a system object) and not the kernel. If the object is available in a local partition, the object manager maps the virtual space of the called object into the address space of the thread, in the place of the calling object, and starts the thread at the entry point in the called object.
Page missing from report
the DSM partition. The DSM partition implements all the protocols (including coherency protocols) necessary for the Distributed Shared Memory function. The DSM server currently runs on Unix and provides service to Clouds machines.

3.2. The Ra Transport Protocol

All communications between machines running Clouds as well as the communications between diskless Clouds machines and storage servers on Unix are handled through the Ra transport protocol or RaTP. RaTP is a connectionless, reliable protocol that efficiently provides client-server communications using the message transaction model. RaTP is a minimal protocol and provides only the functionality necessary for RPC and DSM based object invocations. More functionality will be layered on top of RaTP if necessary. RaTP is quite similar to (but much simpler than) the VMTP protocol used by the V-System [Ch86].

3.3. Programming Support

Programming objects in Clouds v.2. can be done through any structured, procedural language. The linker and the loader have to be modified to generate an object in the proper format, from compiled modules of the language.

However most of the facilities of Clouds will be inaccessible to the programmer using an off-the-shelf language. For this reason, we are experimenting with a pre-processed (and post-processed) version of C++. The features necessary for supporting programming of Clouds objects are:

- Defining an object class from existing classes, or from the default parent class.
- Declaring the entry points.
- Mechanisms for instantiation of object instances from class instances.
- Labeling the entry points with consistency labels (see section on atomicity.)
- Declaring data segments.
- Support for dynamically changing the size of the data segments.
- Access to the object naming functions, capability storage and manipulation.
- Access to the semaphores and locks provided by Clouds.
- Support for creation and deletion of data segments and using data segments to create objects.

In theory, all these can be built into an existing programming language. Some of the features (especially the last three) can be provided by runtime libraries.

The C++ program segments that define a Clouds object is compiled on Unix and a set of Clouds segments are generated. The segments are loaded on the Clouds segment server(s) and can be invoked from Clouds.

Another set of programming tools that will be provided by Clouds are utility objects and predefined objects. Utilities will be similar to utilities in any operating system, and will be provided by objects that will be a part of the Clouds environment. Predefined objects will be object classes which can be instantiated by the user to provide some services that are deemed useful. We are in the preliminary stages of research in this area.

3.4. User Interfaces

We plan to use Unix and X-windows as our interface to Clouds. Unix programs can make use of Clouds facilities through invocation support provided by a Clouds library on Unix. Also, Clouds utilities will be available under X-windows. This will have several implications:

Firstly, Clouds can be treated as a back-end system to the Unix workstation, for distributed processing, computations, object-oriented programs and atomic programs. All these facilities will be available to Unix programs and the user.

Second, the user can access Clouds utilities through the X-window system, and thus
Page missing from report
terleave with other s-threads, as well as with cp-threads.

There are two varieties of cp-threads, namely the gcp-thread and the lcp-thread. The gcp-thread semantics provide global (heavyweight) consistency and the lcp-thread semantics provide local (lightweight) consistency.

All threads are s-threads when created. The handling of cp-threads are programmed by the following scheme. All operations defined on objects in Clouds are tagged with a consistency label; the labels used are:

- Globally-Consistent (gcp)
- Locally-Consistent (lcp)
- Standard (s)
- Inherited (i)

An object can have any number of different labels on its operations. Also, the same operation may have multiple entry-points, labeled at different atomicity levels. A s-thread executing a gcp or lcp operation converts to a gcp or lcp-thread. A thread entering a lcp entry point commits its updates (inside this object) just before it exits the object. This provides intra-object consistency rather than the inter-object consistency provided by the gcp operations, and thus is a cheap method of updating one object atomically. Locking and recovery are automatic.

The standard entry points do not support any locking or recovery. They can make use of "best-effort" semantics. They can also be used for non-traditional purposes such as peeking at incomplete results of actions (as they are not hindered by locking and visibility rules of actions). Locks are available for synchronizing s-threads, but recovery is not supported.

A thread entering an object through an operation with the inherited label simply retains its previous consistency type.

The combination of the consistency labels in the same object (or in the same thread) lead to many interesting (as well as dangerous) execution time possibilities. Especially when s-threads update data being read/updated by gcp or lcp threads. The complete discussion of the semantics, behavior and implementation of this scheme is beyond the scope of this paper, and the reader is referred to [ChDa89].

4.2. Fault Tolerance

Transaction processing systems and systems like Clouds that support consistency of data, provide guarantees about consistency of data if computations do not complete (due to failures). However they do not guarantee success of computations. The following section discusses a subsystem for Clouds that is designed to allow non-stop computations.

This system uses a mechanism called parallel execution threads or PET. The attributes of the system guarantee uninterrupted processing in face of pre-existing (static) failures, as well as system and software failures that occur while a resilient computation is in progress (dynamic failures).

To obtain these property, the basic requirements of the system are:

- Replication of objects, for tolerating static and dynamic failures.
- Replication of computation, for tolerating dynamic failures.
- A quorum-based updating method and coordinated commit mechanism to make the scheme work.

The PET system works by first replicating all the critical objects at different sites in the system. The degree of replication is dependent on the degree of resilience required.

When a resilient computation is initiated, separate replicated actions (gcp-threads), on a number of sites. The number of sites is another parameter provided by the user, and reflects the degree of resilience required. The separate actions (or Parallel Execution Threads) run using DSM invocations on the replicated objects. An invocation by one
Page missing from report
provided by Argus is for the Argus language. Clouds on the other hand is a general purpose operating system, not tied to any language. Currently a modified version of C++ is being used for programming applications for Clouds, but other languages will be added later.

5.2. Eden

Eden is a object-based distributed system, implemented on the Unix operating system at the University of Washington. Eden objects (called Ejects) use the active object paradigm, that is each object consists of a process and an address space. An invocation of the object consists of sending a message to the (server) process in the object, which executes the requested routine, and returns the results in a reply [Alm83, AlB183, NoPr85].

Since every object in the system needs to have a process servicing it, this could lead to too many processes. Thus Eden has an active and a passive representation of objects. The passive representation is the core image of the object stored on the disk. When an object is invoked, it must be active, thus invoking a passive object involves activating it. A process is created by 'exec'-ing the core image of the object (frozen earlier), and then performs the required operation. The activation of passive objects is an expensive operation. Also concurrent invocations of objects are difficult and are handled through multi-threaded processes or coroutines.

The active object paradigm and the Unix-based implementation are some of the major differences between Eden and Clouds. Eden also provides support for transaction and replication objects (called Reflects). The transaction support and replication were added after the basic Eden system was designed and have some limitations due to way Unix handles disk I/O.

5.3. Cronus

Cronus is an operating system designed and implemented at BBN Laboratories. Some of the salient points of Cronus are the integration of Cronus functions with Unix functions, the ability of Cronus to handle a wide variety of hardware and the coexistence of Cronus on a distributed set of machines running Unix, as well as several other host operating systems [BeRe85, GuDe86, ScTh86].

Like Eden, Cronus uses the active objects. This is necessary to be able to make Cronus run on top of most host operating systems. Cronus objects are handled by managers. Often a single manager can handle several objects, by mapping the objects into its address space. The managers are servers and receive invocation requests through catalogued ports. Any Unix process on any machine on the network can avail of Cronus services from any manager, by sending a message to the appropriate manager. By using canonical data forms, the machine dependencies of data representations are made transparent. Irrespective of the machine types, any Unix machine can invoke Cronus objects in a location independent fashion.

5.4. ISIS

ISIS (version 1) is a distributed operating system, developed at Cornell University, to support fault tolerant computing. ISIS has been implemented on top of Unix. It uses replication and checkpointing to achieve failure resilience. If data object is declared to be k-resilient, the system creates k+1 copies of the object. The replicated object invocation is handled by invoking one replica and transmitting the state updates to all replicas. Checkpointing at each invocation is used to recover from failures [Bi85].

The goals and attributes of ISIS are different from Clouds. ISIS is built on top of some interesting communication primitives and is not built as a general purpose computing environment.

5.5. ArchOS and Alpha

Alpha is the kernel for the ArchOS operating system developed by the Archons project at Carnegie Mellon University. Like Clouds, the Alpha kernel is a native operating system kernel designed to run on the special hardware called Alpha-nodes. The Alpha kernel uses passive objects residing in their own virtual spaces, similar to Clouds. ArchOS is designed for real time applications support-
Page missing from report
programming support research; Jose Bernabeu, Yousef Khalidi and Phil Hutto for their efforts in making the v.1 kernel usable and for the design and implementation of Ra; Sathis Menon for significant contribution to the implementation of v.2. as well as managing the software development effort. Also Mustaque Ahmad, Ray Chen, Greg Kenley, Kishore Ramachandran, Henry Strickland and Chris Wilkenloeh for their participation in and contributions to the project.

8. References


[GrSeWe86] J. Greif, R. Seliger and W. Weihl, Atomic Data Abstractions in a Distributed Collaborative
Page missing from report


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

Raymond C. Chen  
Partha Dasgupta

School of Information and Computer Science  
Georgia Institute of Technology  
Atlanta, GA 30332

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 failure, 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] [PN83] [SBD84]) 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 ([LDH87]) 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/Calmeol system ([HW87]) allows users to write their own commit routines to provide custom recovery for similar purposes. Quicksilver ([IMSC88]) 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 ([WLP88]) 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 computations, allowing 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 ([WCC84] [Jon79] [Lis82] [LS83] [All83] [ABL85] [BMS85] [B85]). 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 ([BAHK89]). 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 through 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 kind 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 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.

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

Note: "T → " ≡ "Thread transforms to"

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

Figure 1: Consistency Label Compatibility
Global Consistency Preserving

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 change-set version of that item is created. Further updates to that item by the same thread are reflected in the change-set version.

Let a change-set be the set of the names of all data items of 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, a new shadow version of that item is created. Further updates to that item by the same thread are reflected in the shadow version.

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 for that gcp-thread, the name of the item being updated is added to the change-set. Upon completion of the transforming invocation, the change-set for that operation 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 nested 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 changes made in the shadow version of the data item are reflected in the permanent system state. Instead, those changes are reflected in the change-set of the transforming invocation for that thread.

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.)

Inherited operations are compatible with all threads and indicate the presence 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).

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 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 releases 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 straightforward 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.
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 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 persistent 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.
consistent with the "best-effort" semantics of s-threads. This, however, is not the case if the overwritten data item is system-locked. Thus, those changes are immediately 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 gcps-thread completes. Changes made in thread segment #2 and #4 are simultaneously committed to the stable view or aborted when invocation C 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 the invocations shown in figure 3. Likewise, changes made in thread segment #5 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).

### 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 gcps-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 gcps-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 the invocations shown in figure 3. Likewise, changes made in thread segment #5 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 gcps-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 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 three 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 a 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 ob-
block {
  char data[BLOCKSIZE];
  int block.id;
}

defile {
  segment file_data {
    int numblocks;  // # backed by allocator
    block filedata[];  // real data;
  }
  segment block_data {
    // list of integers
  }
  lcp-entry status read(int, char [BLOCKSIZE]);
  lcp-entry status write(int, char [BLOCKSIZE]);
  lcp-entry status truncate(int);
  lcp-entry status get_blockids(id_list ids);
  try status
  
  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 block_ids
      } else abort();
    strncpy(data[i], buf, BLOCKSIZE);
  }
  return(SUCCESS);
}

deftry status {
  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);
}

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
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, and implementation complexity.

10 Implementation

The semantics presented here, while abstract, naturally lend
themselves to being implemented in and supported by the oper-
sating system. The semantics are a super-set of the basic ob-
ject/thread semantics as defined in section 2, and certain fea-
tures such as the automatic system-level locking would be diffi-
cult 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, com-
mit/abort processing, and system-level locking are being imple-
mented 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 struc-
tures. This allows flexible grouping of variables and data struc-
tures 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

127
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 locked or allowed to proceed, whether a shadow version must be created, and which page table entries should be reset to allow read or write access. A simple access violation handler shown in figure 5. This version shadows only the updated pages in a system-locking segment. A more sophisticated version could attempt to decrease the number of access violations by shadowing other pages around a touched page in anticipation of them being touched later.

1 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 read creation semantics may be used to achieve action/nested-transaction 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.

2 Acknowledgements

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

References


The Clouds Distributed Operating System:

*Functional Description, Implementation Details and Related Work.*

**Partita Dasgupta, Richard J. LeBlanc Jr., & William F. Appelbe.**

School of Information and Computer Science
Georgia Institute of Technology
Atlanta GA 30332

---

**Abstract**

Clouds is an operating system in a novel class of distributed operating systems providing the integration, reliability and structure that makes a distributed system usable. Clouds is designed to run on a set of general purpose computers that are connected via a medium-to-high speed local area network. The structure of Clouds promotes transparency, support for advanced programming paradigms, and integration of resource management, as well as a fair degree of autonomy at each site.

The system structuring paradigm chosen for the Clouds operating system, after substantial research, is an object/thread model. All instances of services, programs and data in Clouds are encapsulated in objects. The concept of persistent objects does away with the need for file systems, and replaces it with a more powerful concept, namely the object system. The facilities in Clouds include integration of resources through location transparency; support for various types of atomic operations, including conventional transactions; advanced support for achieving fault tolerance, and provisions for dynamic reconfiguration.

---

**1. Introduction**

Clouds is a distributed operating system under development. The goal of the Clouds project is to develop an instance of a class of distributed operating systems that provide the integration, reliability and structure necessary to make distributed computing usable.

Clouds is designed to run on a set of general purpose computers (unprocessors or multiprocessors) that are connected via a medium-to-high speed local area network. The major design objectives for Clouds are:

- Efficient design and implementation.
- Simple and uniform interfaces for distributed processing.
- The paradigm used for defining and implementing the software structure of the Clouds system, chosen after substantial research is an object/thread model. This model provides threads to support computation and objects to support an abstraction of storage. (These concepts are defined in sections 2 though 4). This model has been augmented to support atomicity of computation to provide support for reliable programs [Al83, ChDaS7]. In this paper, we provide a functional description of the system (sections 2 to 6), some implementational details (section 7), and discussion of related work (section 9).

**1.1. Current Status**

The first version of the Clouds operating system has been implemented and is operational. This version is referred to as Clouds v.1. This is being used as an experimental testbed by the implementors.

Some of the performance figures for Clouds v.1 were:

<table>
<thead>
<tr>
<th>Type of Invocation</th>
<th>Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Invocations</td>
<td>10</td>
</tr>
<tr>
<td>Remote Invocations</td>
<td>40</td>
</tr>
<tr>
<td>Commit of 1 page data</td>
<td>180</td>
</tr>
</tbody>
</table>

These figures are large due to several factors. The VAX architecture was not very suitable for implementing objects, and flushing of the translation buffers for each invocation causes the local invocation to be more expensive than expected [RaKh88]. The Ethernet hardware used in our VAX-11/750 is slow, and coupled with a non-optimized driver gives us poor performance on round trip messages and hence large remote invocation times. The disk used in the commit tests was also exceedingly slow (40msec seek, 25msec/page write.) However, the experience with this version has taught us that the approach works. It also taught us how to do it better.

The lessons learned from this implementation are being used to redesign the kernel and build a new version. The basic system paradigm, the semantics of objects and threads and the goals of the project remain unchanged and v.2 will be identical to v.1 in this respect.

The structure of Clouds v.2 is different. The operating system will consist of a minimal kernel called "Ra". Ra will support the basic function of the system, that is location independent object invocation. The operating system will be built on top of the Ra kernel using system level objects to provide systems services (user object management, synchronization, naming, atomicity and so on.)
2. Objects

All data, programs, devices and resources on Clouds are encapsulated in entities called objects. The only entity recognized by the system, other than an object, is a thread. A Clouds object, at the lowest level of conception, is a virtual address space. Unlike virtual address spaces in conventional operating systems, a Clouds object is neither tied to any process nor is volatile. A Clouds object exists forever (like a file) unless explicitly deleted. As will be obvious in the following description of objects, Clouds objects are somewhat 'heavyweight', that is they are suited for storage and execution of large-grained data and programs. This is due to the fact that invocation and storage of objects bear some non-trivial overhead.

Every Clouds object is named. The name of an object, also known as its capability, is unique over the entire distributed system and does not include the location of the object. That is, the capability-based naming scheme in Clouds creates a uniform, flat system name space for objects, and allow for object mobility used for load balancing and reconfiguration.

An object consists of a named address space, and the contents of the address space. Since it does not contain a process, it is completely passive. Hence, unlike objects in some object based systems, a Clouds object is not associated with any server process. The first system to use passive objects, though in a multiprocessor system was Hydra ([Wu74, WuLcSI]).

Threads are the active entities in the system, and are used to execute the code in an object (details in sections 2 and 3). A thread executes in an object by entering it through one of several entry points, and after the execution is complete the thread leaves the object. Several threads can simultaneously enter an object and execute concurrently (or in parallel, if the host machine is a multiprocessor.)

Objects have structure. They contain, minimally, a code segment, a data segment and a mechanism for extending limits of storage allocated to the object. Threads that enter an object execute in the code segment. The data segment is accessible by the code in the code segment, but not by any other object. Thus the object has a wall around it which has some well-defined boundaries, though which activity can come in. Data cannot be transmitted in or out of the object freely, but can be moved as parameters to the code segment entry points (see discussion on threads).

Clouds objects can be defined by the user or defined by the system. Most objects are user-defined. Some examples of system-defined objects are device drivers, name-service handlers, communication systems, systems software, utilities, and so on. The basic kernel (Ra) is not an object; it is an entity that provides the support for object invocation. A complete Clouds object can contain user-defined code and data; system-defined code and data that handle synchronization, recovery and commit; a volatile heap for temporary memory allocation; a permanent heap for allocating memory that will remain permanent as part of the data structures in the object; locks; and capabilities to other objects.

Files in conventional systems can be conceived of a special case of a Clouds object. Thus, Clouds need not support a file system, but use an object system. This is discussed in further detail in section 4.

Though Clouds objects can be created, deleted and manipulated individually, the operating system is designed to support a class and instantiation mechanism. An object in the system can be an instance of its template. An object of a certain type is created by invoking a 'create' operation on the template of this type. Each template is created by invoking a create operation on a single template-template, which can create any template, if provided, as argument, the code and data definitions of the template.

The templates, the template-template and all the instances thereof, are regular Clouds objects, and, as discussed earlier, they exist from the time of creation, until explicitly deleted.

3. Threads

The only form of activity in the Clouds system is the thread. A thread can be viewed as a thread of control that executes code in objects, traversing objects as it executes. Threads can span objects, and can span machine boundaries. In fact, machine boundaries are invisible to the thread (and hence to the user). Threads are implemented in the Clouds system as lightweight processes, comprising of a PCB and a stack (but no virtual space). A thread that spans machine boundaries is implemented by several processes, one per site.

Upon creation, a thread starts up at an entry point of an object. As the thread executes, it executes code inside an object and manipulates the data inside this object. The code in the object can contain a procedure call to an operation of another object. When a thread executes this call, it temporarily leaves the caller object and enters the called object, and commences execution there. The thread returns to the caller object after the execution in the called object terminates. The calls to the entry point of objects are called object invocations. Object invocations can be nested. The code that is accessible by each entry point is known as an operation of the object.

A thread executes by processing operations defined inside many objects. Unlike processes in conventional operating systems, the thread often crosses boundaries of virtual address spaces. Addressing in an address space is, however, limited to that address space, and thus the thread cannot access any data outside an address space. Control transfer between address spaces occurs through virtual space. A thread that spans machine boundaries is implemented by several processes, one per site.

When a thread executing in an object (or address space) executes a call to another object, it can provide the called operation with arguments. When the called operation terminates, it can send back result arguments. That is, object invocations may carry parameters in either direction.

These arguments are strictly data, they may not be addresses. Note that names (capabilities) are data. This restriction is necessary as the address space of each object are disjoint, and an addresses is meaningful only in the context of the appropriate object. Parameter passing uses the copy-in-copy-out method.

4. The Object/Thread Paradigm

The structure created by a system composed of objects and threads has several interesting properties.

First, all inter-object interfaces are procedural. Object invocations are equivalent to procedure calls on modules not sharing global data. The modules are permanent. The procedure calls work across machine boundaries. (Since the objects exists in a global name space, there is no user-level concept of machine boundaries.) Although local invocations and remote invocations (also known as remote procedure calls or RPC) are differentiated by the operating system, this is transparent to the application and systems programmers.
Second, the storage mechanism used in the object-based world is quite different from that used in the conventional operating systems. Conventionally, the file is the storage medium of choice for data that has to persist, especially since memory is lost to processes and processes can die and lose all the contents of their memory. However, memory is easier to manage, more suited for structuring data and essential for processing. The object concept merges these two views of storage, and creates the permanent virtual space.

For instance, a conventional file is a special case of an object. That is, a file is an object with operations such as read, write, seek, and so on, defined in it. These operations transport data in and out of the object through parameters provided to the calls.

Though files can be implemented using objects, the need for having files disappear in most situations. Programs do not need to store data in file-like entities, since they can keep the data in the data space in each object, structure appropriately. The need for user-level naming of files transforms to the need for user-level naming for objects.

Just as Clouds does not have files, it does not provide user-level support for file (or disk) I/O. In fact there is no concept of a “disk” or such I/O devices (except user terminals). The system creates the illusion of a huge virtual memory space that is permanent (non-volatile), and thus the need for using disk storage from a programmer’s point of view, is eliminated.

Messages are a paradigm of choice in message-based distributed systems. In the object-thread paradigm, like the need for I/O, the need for messages is eliminated. Threads need not communicate through messages. Thus ports are not supported. This allows a simplified system management strategy as the system does not have to maintain linkage information between threads and ports.

Just as files can be simulated for those in need for them, messages and ports can be easily simulated by an object consisting of a bounded buffer that implements the send and receive operations on the buffer. However, we feel that the need for files and messages are the product of the programming paradigms designed for systems supporting these features, and these are not necessary structuring tools for programming environments.

A programmer’s view of the computing environment created by Clouds is apparent. It is a simple world of named address spaces (or objects). These objects live in computing systems on a LAN, but the machine boundaries are made transparent, creating a unified object space. Activity is provided by threads moving around amongst the population of objects through invocation; and data flow is implemented by parameter passing. The system thus looks like a set of permanent address spaces which support control flow through them, constituting what we term object memory.

This view of a distributed system does have some pitfalls. However these problems can be dealt with using simple techniques (implemented by the system), which are outlined below.

Threads aborting due to errors will leave permanent faulty data in objects they have modified. Failure of computers will result in similar mishaps. Multiple threads invoking the same object will cause errors due to race conditions and conflicts. More involved consistency violations may be the results of non-serializable executions. In a large distributed system, having thousands of objects and dozens of machines, corruption due to failure cannot be tolerated or easily repaired. The prevention of such situations is achieved through the use of atomicity at the processing level (not necessarily atomic actions). The following section gives a brief overview of the atomicity properties supported by Clouds.

5. Atomicity

The action support is an area where the Clouds v.1 and Clouds v.2 differ.

5.1. Actions in Clouds v.1.

In the first design, Clouds supported atomic actions and nested actions somewhat based on the model defined by Moss in his thesis [Mo81]. Clouds v.1 extended Moss’s model by allowing custom tailored synchronization and recovery, as well as interactions between actions and non-actions.

The synchronization and recovery properties can be localized in objects, on a per object basis. The synchronization and recovery can be handled by the system (to adhere to Moss’s semantics) or can be tailored by the user and thus provide facilities beyond those allowed by standard nested transactions. Customization is allowed by labeling of objects as “auto-sync” or “custom-sync” and “auto-recoverable” and “custom-recoverable”. Further details can be found in [Wi87].

5.2. Atomicity in Clouds v.2.

The support for atomicity in Clouds v.2 has its roots in the above scheme, but has been changed in some respects. The following is a brief outline of the scheme. The actual methods used are discussed in greater detail in [ChDw87].

Instead of mandating customization of synchronization and recovery for application that cannot use strict atomicity semantics, the new scheme support a variety of consistency preserving mechanisms. The threads that execute are are of two kinds, namely s-thread (or standard threads) and cp-threads (or consistency-preserving threads). The s-threads have a “best effort” execution scheme and are not provided with any system-level locking or recovery. The cp-threads on the other hand are supported by locking and recovery schemes, provided by the system. When a cp-thread executes, all pages it reads are read-locked and the pages it updates are write-locked. The updated pages are written using a 2-phase commit mechanism when the cp-thread completes.

The data in the system has an instantaneous version and a stable version. In fact, if nested threads are used, the data has a stack of versions, the top being the instantaneous version and the bottom being the stable version. All the threads work on the instantaneous version. The data updated by cp-threads are committed when the cp-thread exits, while the data touched by the s-threads are committed “eventually”, using a best effort semantics.

The cp-threads are allowed to interleave with s-threads, and also the cp-threads can be used to provide heavyweight as well as lightweight atomicity, using gep and lcp operations, described below.

All threads are s-threads when created. The handling of cp-threads are programmed by the following scheme. All operations in objects in Clouds are tagged with a consistency label, the labels used are:
A thread entering a gcp entry point commits its updates (inside this object) as soon as it exits the object. This provides intra-object consistency rather than the inter-object consistency provided by the gcp operations, and thus is a cheap method of updating one object atomically. Locking and recovery systems should be implementable with automatic.

The standard entry points do not support any locking or recovery. They can make use of "best-effort" semantics. They can also be used for non-traditional purposes such as pecking at incomplete results of actions (as they are not hindered by locking and visibility rules of actions). Locks are available for synchronizing non-actions, but recovery is not supported.

The other labels as well as combination of these labels in the same object (or in the same thread) lead to many interesting (as well as dangerous) variations. The complete discussion of the semantics as well as the implementation is beyond the scope of this paper, and the reader is referred to [ChDa87].

6. Programming Support

Systems and application programming for Clouds involves programming objects that implement the desired functionality. These objects can be expressed in any programming language. The compiler (or the linker) for the language, however, must be modified to generate the stubs for the various entry points, invocation handler, system call interfaces and the inclusion of default systems function handling code (such as synchronization and recovery.)

The language Aeolus has been designed to integrate the full set of powerful features that the Clouds kernel supports. Aeolus currently supports the features of Clouds v.1, but is being expanded for added functionality of Ra and Clouds v.2. [LeWi85, WiBi85, WiLe86].

Aeolus is the first generation language for Clouds. It does not support some of the features found in object-oriented programming systems such as inheritance and subclassing. Providing support for these features at the language level is currently under consideration.

7. Implementation Notes

The implementation of the Clouds operating system has been based on the following guidelines:

- Since one of the primary aims of Clouds is to provide the substrate for reliable, fault tolerant computing, the kernel and the operating system should provide adequate support for implementing fault tolerance. (Fault tolerance in not discussed in this paper, the reader is referred to [AhDa87].)
- The system design should be simple to comprehend and implement.

7.1. Hardware Configuration

Clouds v.1 was built on a three VAX-11/750 computers, connected through an Ethernet, equipped with RL02 and RA81 disk drives. The user interface was through the Ethernet, accessible from any Unix machine.

Clouds v.2 will be implemented on a set of Sun-3 class machines. The cluster of Clouds machines will be on an Ethernet, and user will be able access them through workstations running Clouds as well as any Unix workstation.

7.2. Software Configuration and Kernel Structure

The kernel (version 2.) used to support Clouds is called Ra. Ra is a native kernel running on bare hardware. The kernel is implemented in C for portability, and because the availability of C source for the UNIX kernel simplified the task of developing hardware interfaces such as device drivers.

The kernel runs on the native machine and not on top of any conventional operating system for two reasons. Firstly, this approach is efficient. As Clouds does not use much of the functionality of conventional operating systems (such as file systems), building Clouds on top of a Unix-like kernel make poor use of the host operating system. Secondly, the paradigms and the support for synchronization, recovery, shared memory and so on; used in Clouds are considerably different from the functionality provided by conventional operating systems, and major changes would be necessary at the kernel level of any operating system in order to implement Clouds.

The Ra kernel provides support for partitions, segments, virtual spaces, processes and threads. These are the basic building blocks for Clouds. The partitions provide non-volatile storage, the segments provide memory storage, which are used to build objects, which in turn reside in virtual spaces. Processes provide activity which are used to compose threads. A description of the design of Ra can be found in [BeHuKh87].

7.3. Object Naming and Invocation

The two basic activities inside the Ra kernel are system call handling and object invocations. System call handling is done locally, as in any operating system. The system calls supported by the Ra kernel include object invocation, memory allocation, process control and synchronization, and other localized systems functions. Object invocation is a service provided by the kernel for user threads. The attributes that object invocation satisfy are:

- Location independence.
- Fast, for both local and remote invocations.
- Failed machines should not hamper availability of objects on working sites, from working sites.
- Moving objects between sites, reassigning disk units and so on should be simple (for reconfiguration and fault tolerance support).
Location independence is achieved through a capability based naming system. Availability is obtained through decentralization of directory information and a search-and-invocation strategy coupled with a multicast based object location scheme, designed for efficiency [ALAm87]. Speed is achieved by implementing the invocation handlers at the lowest level of the kernel, on the native machine.

7.4. Storage Management

The storage management system handles the function required to provide the reliable, permanent object address spaces. As mentioned earlier, unlike conventional systems, where virtual address spaces are volatile and short-lived, Clouds virtual spaces contain objects and are permanent and long-lived. The first version of the implementation is detailed in [P86].

The storage management system stores the object representations on disk, as an image of the object space. When an object is invoked, the object is demand paged into its virtual space as and when necessary. As the invocation updates the object, the updated pages do not replace the original copy, but have shadow copies on the disk. The permanent copy is updated only when a commit operation is performed on the object. The storage manager provides the support to commit an object using the two-phase commit protocol.

7.5. User Interfaces

User interfaces can make or break an operating system. Users do not like to switch systems, and have to re-learn the interfaces. We plan to use Unix and X-windows as our interface to Clouds. Unix programs can make use of Clouds facilities through invocation support provided by a Clouds library on Unix. Also, Clouds utilities will be available under X-windows. This will have several implications:

Firstly, Clouds can be treated as a back-end system to the Unix workstation, for distributed processing, computations, object-oriented programs and atomic programs. All these facilities will be available to Unix programs and the user.

Second, the user can access Clouds utilities through the X-window system, and thus making the learning time much smaller. We believe this approach will make Clouds easier to access and use, and we hope to build a large user community that is essential to the success of new operating systems.

8. Comparisons with Related Systems

Clouds is one of the several research projects that are building object-based distributed environments. Although there are differences between all the approaches, we feel that the area of distributed operating systems is not mature enough to conclusively argue the superiority of one approach over the other. In the following paragraphs we document the major differences between Clouds and some of the better known projects in distributed systems. (This list is not exhaustive).

One of the major differences between Clouds and some of the systems mentioned below is in the implementation of the kernel. Many systems implement the kernel as a Unix process, while Clouds is implemented as a native operating system (as are Mach and Alpha). Clouds is not intended to be an enhancement, or replacement of, the UNIX kernel. Instead, Clouds provides a different paradigm from that supported by UNIX (e.g., the UNIX paradigms of "devices as files", unstructured files, volatile address spaces, pipes, redirection etc.)

8.1. Argus

Argus is a language for describing objects, actions and processes using the concept of a guardian. The language defines a distributed system to be a set of guardians, each containing a set of handlers. Guardians are logical sites, and each guardian is located at one site, though a site may contain several guardians. The handlers are operations that can access data stored in the guardian. The data types in Argus can be defined to be atomic, and atomic data types changed by actions are updated atomically when the action terminates [WeLi83, LiSc83]. The support for Argus is built on top of Unix, and provides all the facilities of the Argus language [Li87].

Some of the similarities between Argus and Clouds are in the semantics of nested actions. Both use the nested action semantics and locking semantics that are derived from Moss. This includes conditional commit and lock inheritance. However the consistency preserving mechanisms in Clouds have moved away from Moss's action semantics, substantially, though retaining the nested action semantics as a subset. Also the guardians and handlers in Argus have somewhat more than current similarities to objects in Clouds, as the design of Clouds was influenced by Argus.

The differences include the implementation strategies, programming support and support for reliability. The scheme of permanent virtual spaces provided by passive objects is a major difference. As mentioned earlier, Argus is implemented on top of a modified Unix environment. This is one of the reasons for the somewhat marginal performance of the Argus system observed in [GrSeWe86]. The programming support provided by Argus is for the Argus language. Clouds on the other hand is a general purpose operating system, not tied to any language. Though Aeolus is the preferred language at present, we have used C extensively for object programming. We have plans to implement more object-oriented languages for the the Clouds system.

8.2. Eden

Eden is an object-based distributed system, implemented on the Unix operating system at the University of Washington. Eden objects (called Ejects) use the active object paradigm, that is each object consists of a process and an address space. An invocation of the object consists of sending a message to the (server) process in the object, which executes the requested routine, and returns the results in a reply [Alm83, AlB83, NoPr85].

Since every object in the system needs to have a process servicing it, this could lead to too many processes. Thus Eden has an active and a passive representation of objects. The passive representation is the core image of the object stored on the disk. When an object is invoked, it must be active, thus invoking a passive object involves activating it. A process is created by 'exec'-ing the core image of the object (frozen earlier), and then performs the required operation. The activation of passive objects is an expensive operation. Also concurrent invocations of objects are difficult and are handled through multithreaded processes or coroutines.

The term kernel has been used quite frequently to describe the core service center of a system. However, when this service is provided by a Unix process rather than a resident, uninterruptible memory, the usage of the term is somewhat counter-intuitive.
The active object paradigm and the Unix-based implementation are some of the major differences between Eden and Clouds. Eden also provides support for transaction and replication objects (called Replicates). The transaction support and replication were added after the basic Eden system was designed and have some limitations due to manner Unix handles disk I/O.

### 8.3. Cronus

Cronus is an operating system designed and implemented at BBN Laboratories. Some of the salient points of Cronus are the integration of Cronus functions with Unix functions, the ability of Cronus to handle a wide variety of hardware and the existence of Cronus on a distributed set of machines running Unix, as well as several other host operating systems. Clouds as well as several other host operating systems.

Lin to be chic to make Cronus run on top of most host operating systems. Cronus objects are handled by managers. Often a single manager can handle several objects, by mapping the objects into its address space. The managers are servers and receive invocation requests through catalogued ports. Any Unix process on any machine on the network can avail of Cronus services from any manager, by sending a message to the appropriate manager. By use of canonical data forms, the machine dependencies of data representations are made transparent. Irrespective of the machine types, any Unix machine can invoke Cronus objects in a location independent fashion.

### 8.4. ISIS

ISIS (version 1) is a distributed operating system, developed at Cornell University, to support fault tolerant computing. ISIS has been implemented on top of Unix. It uses replication and checkpointing to achieve failure resilience. If a data object is declared to be k-resilient, the system creates k+1 copies of the object. The replicated object invocation is handled by invoking one replica and transmitting the state updates to all replicas. Checkpointing at each invocation is used to recover from failures.

The goals and attributes of ISIS are different from Clouds. ISIS is built on top of some interesting communication primitives and is not built as a general purpose computing environment.

### 8.5. ArchOS and Alpha

Alpha is the kernel for the ArchOS operating system developed by the Archons project at Carnegie Mellon University. Like Clouds, the Alpha kernel is a native operating system kernel designed to run on the special hardware called Alpha-nodes. The Alpha kernel uses passive objects residing in their own virtual spaces, similar to Clouds. ArchOS is designed for real time applications supporting specialized defense related systems and applications.

The key design criteria for ArchOS and Alpha are time critical computations and rather than reliability. Fault tolerance is handled to an extent using communication protocols. Real time scheduling has been a major research topic at the Archons project.

### 8.6. V-System

The V operating system has been developed at Stanford University. V is a compromise between message-based systems and object-based systems. The basic core of V provides lightweight processes and a fast communications (message) system. V message semantics are similar to object invocations in the sense that the messages are synchronous and use the send/reply paradigm. The relationship between processes conforms to the client-server paradigm. A client sends a request to the server, and the client blocks until the server replies.

V allows multiple processes to reside in the same address space. Data sharing is through message passing, though shared memory can be implemented through servers managing bounded buffers. The design goals of V are primarily speed and simplicity. V does not provide transaction and replication support. These can be implemented, if necessary at the application level.

The radical difference between V and Clouds is the paradigm used by Clouds.

### 8.7. Mach

Mach is a distributed operating system under development at Carnegie Mellon (Ac86). Mach maintains object-code compatibility with Unix. Mach extends the Unix paradigms by adding large sparse address spaces, memory mapped files, user provided backing stores, and memory sharing between tasks. Mach is implemented on a host of processors including multiprocessors.

The execution environment for a Mach activity is a task. Threads are computation units that run in a task. A single thread in a task is similar to a Unix process. Ports are communication channels, supporting messages which are typed collection of data objects. In addition, Mach supports memory objects, which are collections of data objects managed by a server.

Support for transactions are not built into Mach, but can be layered on top of Mach and has been implemented by Camelot and Avalon (He87).

The approaches used by Mach and Clouds are fundamentally different, as with V and Clouds.

### 9. Concluding Remarks

Clouds provides an environment for research in distributed applications. By focusing on support for advanced programming paradigms, and decentralized, yet integrated, control, Clouds offers more than 'yet another Unix extension/look-alike'. By providing mechanisms, rather than policies, for advanced programming paradigms, Clouds provides systems researchers an adaptable, high-performance, 'workbench' for experimentation in areas such as distributed databases, distributed computation, and network applications. By adopting 'off the shelf' hardware, the portability and robustness of Clouds are enhanced. By providing a 'Unix gateway', users can make use of established tools. The gateway also relieves Clouds from the necessity of providing emulating services such as provided by Unix mail and text processing.

The goal of Clouds has been to build a general purpose distributed computing environment, suitable for a wide variety of user communities, both within and outside the computer science community. We are striving to achieve this through a simple model of a distributed environment with facilities that most users...
would feel comfortable with. Also we are planning to experiment with increased usage of the system by making it available to graduate courses, and hope the feedback and the criticism we receive from a large set of users will allow us to tailor, enhance and perhaps redesign the system to fit the needs for distributed computing, and thus give rise to wider usage of distributed systems.

10. Acknowledgements

The authors would like to acknowledge Martin McKendry and Jim Allchin for starting the project and designing the first version of Clouds. Gene Spafford and Dave Pits for the implementation, Jose Bernabeau, Yousef Khalidi and Phil Hutto for their efforts in making the kernel usable and for the design of Ra. Also Mustaque Ahamad, Ray Chen, Kishore Ramachandran and Henry Strickland for their participation in the project.

11. References


The *Five Color* Concurrency Control Protocol:  
Non-Two-Phase Locking in General Databases.

*Partha Dasgupta*₁ & *Zvi M. Kedem*₂

**Abstract**

Concurrency control protocols based on two-phase locking are a popular family of locking protocols that preserve serializability in general (unstructured) database systems. This paper presents a concurrency control algorithm (for databases with no inherent structure) that is practical, non-two-phase and allows varieties of serializable logs not possible with any commonly known locking schemes. All transactions are required to predeclare the data it intends to read or write. Using this information, the protocol anticipates the existence (or absence) of possible conflicts, and hence can allow non-two-phase locking.

It is well known that serializability is characterized by acyclicity of the serializability graph representation of interleaved executions. The two-phase locking protocols allow only forward growth of the paths in the graph. The *Five Color* protocol allows the serializability graph to grow in any direction (avoiding two-phase constraints) and prevents cycles in the graph by maintaining transaction access information in the form of data-item markers. The read and write set information can also be used to provide relative immunity from deadlocks.

**Categories and Subject Descriptors:** H.2.4[Database Management]: Systems -- concurrency, transaction processing.  
**General Terms:** Concurrency Control.  
**Additional Key Words and Phrases:** serializability, locking.

---

This research was partially supported by NSF under grants MCS 81-04882, MCS 81-10097, CCR-89 ONR under contract number N0014-85-K-0046, and NASA under contract NAG-1-430.

1: School of Information and Computer Science  
Georgia Institute of Technology  
Atlanta, GA 30332

2: Department of Computer Science  
Courant Institute of Mathematical Sciences  
New York University  
New York, NY 10012
1. Introduction

In this paper we present a locking protocol for database concurrency control, that applies to general databases. The locking strategy is non-two-phase. The protocol uses five kinds of locks. The five categories of locks are read-locks, write-locks, intent-locks and two types of marker locks. We use five colors to assign mnemonic names to these locks.

The protocol requires each transaction to predeclare the data it intends to read or write. This can be achieved by data analysis by the query compiler. The predeclared readset and writeset need not be the exact read/write sets, but can be a superset of the actual sets. The performance however depends upon the closeness of the predeclared sets and the actual sets.

Unlike the two-phase locking protocol, the Five Color protocol uses early release of read locks and late acquisition of write locks to enhance concurrency. The early release of read locks makes this protocol violate the two-phase locking rule. This feature however has to be closely controlled, as it can cause non-serializable behavior. It is widely known that two-phase locking "is in a sense, the best that can be formulated" (Ullman in [U182] pg. 380). The optimality of two-phase locking implies that in the absence of any information about the transactions or the database, all locking protocols must be two-phase [Ya82]. The Five Color protocol allows non-two-phase locking by keeping track of transaction ordering using the predeclared read and write sets; and by addition of a check called validation. We first present a brief introduction to the concepts of serializability, the model of a multiuser database, the factors that limit concurrency in two-phase locking and some related work. Section 2 contains a comprehensive description of the Five Color protocol, including an intuitive description of how it functions and why it ensures serializability. Section 3 explains the formal properties of the protocol and derives a proof of correctness and section 4 outlines a modification to the protocol. Sections 5 and 6 deal with deadlocks and livelocks and Section 7 discusses performance issues.

1.1. Serializability

A database is viewed as a collection of data items, which can be read or written by concurrent transactions. Interleaving of updates can leave the database in an inconsistent state. A sufficient condition to guarantee correctness of concurrent database access is serializability of the actions (reads or writes) performed by the transactions on the data items. That is, the interleaved execution of the transactions should be equivalent to some serial execution of the transactions [BeGo80, RoStLe78]. In this paper, we assume serializability to be the criterion of correctness.

Locking of data items is one of the methods of achieving consistency in the face of concurrent updates. For databases with no inherent structure (e.g. databases not organized as DAG's, trees, etc.) the two-phase locking protocol is the most popular locking protocol. However, two-phase locking is restrictive with respect to the amount of concurrency it allows.
Informally, a log is a sequence of actions issued by various transactions on several data items in the database. The transaction actions may be interleaved with one another. Serializability is a syntactic property of a log. It has been shown that recognizing serializability is an NP-complete problem [PaBeRo77, Pa79]. The NP-completeness of the serializability recognition problem implies that we cannot have a scheduler that allows all serializable logs and disallows non-serializable ones, and works in polynomial time (unless P=NP). However certain subclasses of serializable logs are efficiently recognizable in polynomial time. Efficient algorithms can be built that control the actions of transactions, to ensure that the logs produced by a set of transactions fall into one of these easily recognizable classes of serializability. The two-phase locking protocol is one such algorithm which produces a class of polynomially recognizable serializable logs, namely the two-phase locked logs.

1.2. The Model

A database $D$ is a set of distinct data items $\{x_1, x_2, \ldots, x_n\}$. A transaction system $T$ is a set of transactions $\{T_1, T_2, \ldots, T_n\}$ that operate on the database. The readset (writeset) of a transaction $T_i$ is the set of all items $T_i$ reads (writes).

A transaction that intends to read (or write) a data item $x$, issues a read (or write) request to the transaction manager. The transaction manager is responsible for determining whether or not granting of the request may cause a violation of the correctness criterion (generally serializability). The transaction manager then takes appropriate action by granting, rejecting or delaying the request.

A trace of a transaction is a sequence of successful read and write requests it makes to the transaction manager. A trace is written as a sequence of actions of the form $R_i(x)$ or $W_i(x)$, where $R_i(x)$ (or $W_i(x)$) means a transaction $T_i$ issues a read (write) on data item $x$. Note that we are not interested in the values read or written, but in the syntactic properties of the string of reads and writes on the data items.

We will assume at most one read and at most one write per data item in any trace. If a transaction reads as well as writes a particular data item, we assume the read will precede the write. Multiple reads and writes are handled in an obvious way: The first read is used to read the value of the data item and store it in local storage, and the other reads on the same data item are processed locally. Similarly, all writes except the last one are written to local storage, and the last one appears on the log. Thus there is no loss of generality.

A log of a transaction manager is a sequence of reads and writes granted by the transaction manager. As an example, three transaction traces and one possible transaction manager log are depicted below: (the notation is from Bernstein et al. [BeGo80].)

$$T_1 : R_1(x) \ R_1(y) \ W_1(y)$$
$$T_2 : R_2(y) \ W_2(y)$$
Increasing Concurrency

In database concurrency control we are interested in protocols that maximize concurrency and work efficiently. All known concurrency control protocols restrict the logs to a subset of the all possible serializable logs. Two phase locking has been deemed to be quite restrictive since, intuitively, it holds locks for "longer than necessary". That is, after a data item is read, the lock on it is held till no other locks will be necessary (typically until commit). Indeed this is necessary for providing serializability if no other mechanisms other than locking the data items accessed are being used to control concurrency. In addition, a two-phase locking protocol can cause deadlocks, which further degrade its performance.

We contend that this seemingly restrictive nature of two-phase-locking arises from the fact that the protocol does not assume any \textit{a priori} knowledge of the intentions of the transactions. \textit{A priori} knowledge of read and write sets can be used deadlock prevention in two phase protocols. We use \textit{a priori} knowledge to allow non-two-phase executions, in particular:

- Allowing read locks to be released as soon as the data is read.
- Allowing reading of (some) write locked data items.
- Preventing deadlocks due to lock aquisition (another form of deadlock is possible, see section 5.)

The following (trivial) example illustrates the restrictive nature of two-phase locking, and shows how added information can be used to remove some of the restrictions:

\begin{align*}
T_1 & : R_1(x), W_1(x), \ldots, R_1(y), W_1(y) \\
T_2 & : R_2(x) \\
T_3 & : R_2(y)
\end{align*}

In this case, \(T_2\) (and \(T_3\)) reads the value of \(x\) (and \(y\)), and does nothing else. Thus \(T_2\) and \(T_3\) can read \(x\) (or \(y\)) between any two actions of \(T_1\), and still produce a correct serializable execution sequence. However if two-phase locking is used, then \(T_2\) (or \(T_3\)) is restricted to read \(x\) (or \(y\)) at only certain points, depending upon the locking sequence. For example suppose \(T_1\) uses the locking sequence shown in Figure 1. (\(LS(x)\) denotes setting a shared lock on \(x\), \(LX(x)\) denotes setting an exclusive lock on \(x\), and \(U(x)\) denotes unlocking of \(x\).)
Figure 1: A locking sequence used by transaction T₁.

This locking sequence prevents T₂ from reading x, between W₁(x) and R₁(y). However T₁'s locking sequence can be changed so that T₂ can read x between W₁(x) and R₁(y), but then T₃ will not be allowed to read y between W₁(x) and R₁(y).

Since we know that T₂ and T₃ read x and y and do nothing else, we could allow T₂ and T₃ to read x and y interleaved between any steps of T₁ [GaWi82]. The situation would be different if T₂ or T₃ accessed or updated some other data items after reading x or y. However the two-phase locking protocol does not rely upon, or have access to, information about the complete data access patterns of the transactions. We will show how data access information can be used by concurrency control protocols to deal with situations like the one shown above.

The above example also shows that locking sequences of two-phase locked transactions affect the amount of concurrency, depending on the transaction mix. A particular locking sequence in fact may favor one transaction over another. Since it is impossible to predict which transactions will run concurrently, it is not easy to choose locking sequences. In fact, it is commonly believed that, locking should not be handled by transactions or application programs, but should be the responsibility of lower level, consistency preserving routines i.e. the transaction manager.

In order to make locking transparent, practical two-phase locking schemes use read and write requests to obtain locks [see 2V2PL section 1.4]. A read or write request on an
unlocked data item causes the lock to be obtained. All locks are released only when the transaction terminates. Thus it is intuitively clear that the locks are held for extended periods.

In the following sections, we propose a concurrency control protocol that, in cases like the above, would allow $T_2$ and $T_3$ to read $x$ and $y$ interleaved between any action of $T_1$. This would be achieved by either early release of read locks, or by allowing reading of (older values) of write locked data. The locking is handled entirely by the transaction manager. Holding of read locks on read only items is minimized. The protocol is inherently non-two-phase, and is relatively immune from deadlocks.

1.4. Related Work

Database concurrency control has been an active area of research, and has resulted in the development of many protocols for achieving serializability. These basic mechanisms used by the protocols are locking, timestamps and multiple versions [BeGo80, BeGo82, BeHaGo87, Ul82]. A synopsis of the concurrency control methods is beyond the scope of this paper.

A family of protocols called two-version protocols use "before" and "after" values of the data for concurrency control. The Five Color protocol has a similarity with some of them. Also, some protocols use information about the transaction or the database to enhance concurrency. The Five Color protocol uses information about the transactions (in the form of predeclared read and write sets). The following paragraphs contain brief descriptions of protocols in these two broad families.

Bayer, Heller and Reiser [BaHeRe80] presents a protocol where the reads from a transaction are never delayed, and are granted immediately. They use three kinds of locks namely Read, Analyze and Commit locks. The Read locks are compatible with all the other locks, thus readers can see the before value of any transaction that is writing to a data item and has an Analyze lock. Readers can read while the writer is committing, and a graph is maintained to assure they either read all committed versions or all before versions.

A protocol similar to the Bayer et. al. protocol is described in [BeHaGo87]. This protocol is called the two-version two-phase locking protocol (2V2PL). Under this protocol, the transaction manager uses read and write requests from the transactions to set locks. A read request causes a shared read lock to be obtained. A write request causes a write lock to be obtained. (If the transaction already possesses a read lock in the item, the lock is upgraded to a write lock.) The write lock is compatible with other read locks, but not with a write lock. Reader's reading the write locked item, gets to see the older (before) value. After the transaction commits, the write locks are upgraded to an exclusive "Certify" lock which is incompatible with other locks. The incompatibility of the Certify lock does away with the need for maintaining a dependency graph, and introduces a slight delay for transactions that want to read an item that is in the process of being committed.
In both the protocols, the locking is two-phase; that is, the readers hold read-locks until they commit and writers cannot commit until all the read locks on the items they are updating are released. This causes writing transactions to wait for the readers to terminate. This can also cause writer starvation when the read traffic is high. The Five Color protocol is designed to avoid this situation, as all the read-locks are released as soon as the reads are over, and writers do not have to wait to commit. Also upgrading of locks lead to "trivial" deadlocks [GrHoKoOb81, Ko83]. That is if two transactions hold read locks on a data item and both try to upgrade it to write, then the situation is a deadlock. The Five Color Protocol allows lock upgrading, but avoids trivial deadlocks. Of course these features are possible only because of the read-write set predeclaration.

Steams and Rosenkrantz [StRo81] describe a set of actions and conditions that can be used to control concurrency when using before-values. For each conflict they define a set of "actions" that can be taken (non-deterministically). They also define a set of conditions for proper consistency. Some of the actions may cause an abort or rollback at a later stage. These sets of rules gives rise to a family of protocols, and the performance is dependent upon the choice of actions taken in the particular execution. The concurrency control can be thought of as a non-deterministic table driven mechanism. The tables show the actions for each case of conflict. The cases of conflict are: a younger transaction reading a before value of an older transaction; an older transaction reading the before value of a younger transaction and so on. The actions are to allow the read or write, abort one of the transaction or delay the requester. Some of the actions have restrictions as to the "phase" of the requester, and in some cases one of many actions can be chosen. Timestamps are used to decide the older-younger relationships.

It has been shown that some available information about the transactions or the database can be used for increasing concurrency. For instance Kedem and Silberschatz have developed the tree and DAG (directed acyclic graph) protocols that can be used on databases structured like a tree or a DAG, respectively [SiKe80, KeSi82]. These protocols allow non-two-phase locking, and provide higher concurrency than two-phase locking for transactions that traverse the tree or the DAG. In the DAG protocol a transaction is allowed to lock a child if it has locks on the majority of its parents (except for the first node locked), and unlocking may be done in any order. Thus the transaction can access only those data items that form a rooted subgraph of the original graph. These and similar protocols can be used in specialized applications and have practical limitations, but have received substantial theoretical interest.

If the writeset of a transaction is known in advance, timestamp protocols can be made abort free (or progressive) [BuSi83]. This can provide significant improvement in performance as aborts cause severe limitations of throughput in timestamp based systems.

Semantic knowledge about transaction actions has also been used to speed up transaction processing by Garcia-Molina [Ga83]. A partial loss of serializability can be tolerated in some applications and can be used for better performance [FiMi82]. This may not be of interest in
general purpose databases, where consistency is a major issue.

Read and write sets are used to control concurrency in SDD-1 (A System for Distributed Databases) [BeShRo80]. Transactions are divided into classes depending upon their read and write sets and then conflict graph analysis is performed. This is a static classification and is used to determine the nature of the conflicts. The conflict type is used to determine which protocol to use. (SDD-1 uses different protocols for different situations.) SDD-1 is a timestamp based system. In our approach the usage of read and write set information is dynamic. The protocol uses locking and static analysis of conflicting transactions is not performed.

2. The Five Color Protocol

The Five Color Protocol is a non-two-phase locked protocol that ensures serializability in general (unstructured) databases. It derives its name from the five types of locks it uses.

A transaction $T$ acts upon a set of data items $D$. A data item $x \in D$ is in the readset ($Rd$) of a transaction $T$ (that is $x \in Rd(T)$) if the transaction intends to read $x$. Any item that the transaction $T$ intends to write is contained in the writeset ($Wr$) of $T$. Note that $Rd(T)$ need not be a subset of $Wr(T)$ or vice versa, and $Rd(T)$ and $Wr(T)$ may be supersets of the data items actually read and written by the transaction $T$.

Each transaction is required to declare its readset and writeset to the transaction manager before it issues any actions. The read and write sets can be parameters to the begin-transaction statement that is executed by a transaction when it starts. Since the readset (and writeset) are allowed to be supersets of the data items actually read (and written) by the transaction, they could be statically determined during query compilation. Sometimes the transaction may read and write different sets of data depending upon some statically undeterminable conditions. In this case the declared read and write sets should include all the items the transaction may act upon. However, for better performance the difference between the declared and actual read and write sets should be small. Also, the Five Color Protocol transaction manager reads all the items in the readset of a transaction, and thus will cause extra reads if the readset is larger than necessary.

2.1. The Basic Algorithm

After the transaction manager knows the read and write sets of the transaction, it can obtain shared locks on all the data items in the readset, read them and store the values in local storage. Then we would like to release the shared locks, as the locks seem no longer necessary. Also, we would also like to keep the data items in the writeset locked by a shared (intend-to-write) lock while the transaction is running, to prevent other concurrent transactions from updating them and causing missing updates. The intend-to-write lock should be shared, as we would like other concurrent transactions to be able to read the data item before it is actually written. The shared locks on the writeset can then be upgraded to exclusive locks at commit.
time, and the values actually updated.

Thus our protocol is along the following lines:
- Get shared read locks on the read-only items, and shared intend-to-write locks on the write set.
- Read the readset into local storage and release the locks on the read-only items.
- Service the reads and writes issued by the transaction from and to local storage.
- Upgrade the shared locks on the writerset to exclusive locks and perform actual writes to the database during commit, and finally release all locks.

As mentioned earlier, the merits of this approach are the early release of shared read locks and short holding period of exclusive locks. The protocol is obviously non-two-phase, as some shared locks are released before some other shared locks are upgraded to exclusive locks. The problem with this initial scheme is that it can produce non-serializable schedules, and thus is unacceptable.

Now our aim is to use this basic idea, as described above, and add some checks to ensure serializability. We show that this is possible if we use some marker locks and a validation phase. The algorithms used to handle the marker locks are non-trivial and are described in detail in the following sections.

<table>
<thead>
<tr>
<th>old → new</th>
<th>WHITE</th>
<th>BLUE</th>
<th>GREEN</th>
<th>YELLOW</th>
<th>RED</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHITE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>BLUE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>GREEN</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>YELLOW</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>RED</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Figure 2: Lock Compatibility Table (Note: The table is asymmetric).

2.2. Locking

The Five Color Protocol uses five types of locks, namely, Green (GL), Yellow (YL), Red (RL), White (WL) and Blue (BL). The compatibility matrix for these locks are shown in Figure 2.
The **Green** lock is a shared lock used for reading the read-only part of the readset. The **Yellow** lock is a shared intend-to-write lock and is stronger than the **Green** lock. It is used to lock the items in the writeset, as a preparatory measure, before they are actually updated. The **Yellow** lock is compatible with the **Green** locks, allowing a **Yellow** locked item to be read as a read-only data item by another transaction, but it is not compatible with itself, preventing simultaneous update attempts. (The *Five Color* protocol allows upgrading of **Yellow** to **Red**, but not **Green** to **Yellow** or **Red**.)

Note that a **Green** lock can be obtained on a **Yellow** locked item but a **Yellow** lock cannot be obtained on a **Green** locked item. This feature makes the compatibility matrix asymmetric. This asymmetry is not a major feature of the algorithm but has to be provided to prevent a particular race condition that can arise in the lock acquisition phase, described later.

The **Red** lock is the exclusive lock used for writing, and is compatible only with the **White** and **Blue** marker locks. Neither the **Green** locks nor the **Red** locks are held over extended lengths of time. Only **Yellow**, **Blue** and **White** locks exist nearly as long as the transaction does.

The **White** and **Blue** locks are the marker locks. They are compatible with all other locks. These are used by transactions to keep track of data items read or written by other transactions and cause triggering as described later [Ca81]. Briefly, the semantics of the **White** and **Blue** locks are:

- A data item x is **White** locked by a transaction T, if T has read x, or if there is a transaction T’, such that T’ must follow T in a serialization of the execution, and T’ has read x.
- A data item x is **Blue** locked by a transaction T, if there is a transaction T’, such that T’ must follow T in a serialization of the execution, and T’ intends to write x.

### 2.3. Transaction Phases

A transaction T goes through several phases. When the transaction is initiated, *(arrival point)*, the transaction manager obtains **Yellow** locks on the data items in Wr(T) and **Green** locks on Rd(T)–Wr(T) i.e. the read-only data items. This is the **lock acquisition phase**.

After all the locks are obtained, the transaction is considered for **validation**. If the transaction passes validation then it has to acquire some **Blue** and **White** locks. It gets **Blue** (and **White**) locks on data items written (and read) by some other concurrently running transactions. It also has to assign **White** and **Blue** locks on the items in its read and write sets to some other

---

The reason for calling the **Blue** and **White** locks, "locks" and not just "markers" are as follows. Markers are used by agents to mark objects. Irrespective of how many times an object is marked, it becomes unmarked when the marker is removed. With a lock we can ask questions such as *Which agents hold locks on this object?* or *What are the objects locked by this agent?* Conceptually markers are too weak to answer both these questions.
transactions. This is called lock inheritance phase, and the exact details of which data items are locked are explained later. After completion of the lock acquisition phase, the transaction reaches its locked point.

Subsequently all the items in Rd(T) are read into local storage, the Green locks are converted (downgraded) to White locks and the transaction enters the processing phase. Then the transaction commences execution (start point). When the transaction completes execution all Yellow locks are converted to Red locks. This is the Final Locked Point. the updated items are written to the database, all locks are released, and the transaction terminates. These phases and points are illustrated in Figure 3.

The following is an informal outline of how a transaction manager handles a transaction.

→ Arrival Point (Transaction T arrives)

- Get Yellow locks on Wr(T) and compute Before/After sets (explained later)
- Get Green locks on Rd(T) – Wr(T)
- Do validation and lock inheritance processing (explained later)

→ Initial Locked Point

- Read values of Rd(T) into local storage,
- Downgrade Green locks to White locks,
- Start transaction processing.

→ Start Point

- Let T commence processing,
  - if T issues read(x), then return the value of x from local storage,
  - if T issues write(x), then write x in local storage.

- Processing ends.

- Upgrade Yellow locks to Red locks,

→ Final Locked Point

- Write updated items to the database,
- Release all (White, Blue, and Red) locks held by T.

→ Termination Point (Transaction T terminates)
2.4. Transaction Manager Algorithms

A transaction is said to be live if it has arrived, but has not terminated. For each live transaction T, the Transaction Manager maintains two temporary sets, during lock acquisition phase, called Before(T) and After(T). These sets are accessed and updated only during the lock acquisition phase. The set Before(T) is a set of transactions that are live, conflict with T, and must come before T in a serialization order. Similarly, After(T) contains those live and conflicting transactions that must come after T in the serialization order. The serialization order is determined by the actual order in which the locks are requested by the concurrent transactions. (Sometimes the Before and After sets contain a few recently terminated transactions, but that is of no consequence.)

As the transaction manager acquires locks on behalf of a transaction, it can determine which transactions must come before or after this transaction in the serialization order, by looking at the existing locks. These transactions are placed in the Before(T) and After(T) sets. The Before(T) and After(T) sets are constructed as follows.

Suppose T wants a Green lock on a data item x, and a set of transactions \{T_i, T_j, ..., \} already possess Blue locks on x. The existence of Blue locks held by \{T_i, T_j, ..., \} implies that some transaction(s) later than all of \{T_i, T_j, ..., \} have written x. As T wants to read x, it must come after all of \{T_i, T_j, ..., \}. Thus \{T_i, T_j, ..., \} must logically precede T, and they are added to Before(T).

However, if some transaction T_i is holding a Yellow lock on x (when T is trying to Green lock it), this implies T_i will update x after T reads it. Hence T_i should come after T, and T_i is added to After(T).

Similarly, during an attempt to Yellow lock an item x, all transactions holding Blue or White locks on x are added to Before(T).

Validation is simply checking whether Before(T) ∩ After(T) = \emptyset. If not, this implies that there are transactions that must come before as well as after T in the serialization order, and thus the resulting execution could be non-serializable. To prevent this from occurring, the transaction T is rescheduled. Avoiding livelocks (or starvation) due to rescheduling is dealt with in section 6.

If the transaction passes validation, then the lock inheritance processing has to be done. Some Blue and White locks are granted to various transactions as a result of lock inheritance. This is done as follows.

Suppose T has passed validation. For each transaction (τ) in After(T), T is delayed until τ has reached locked point, and T is given White locks on all the data items White or Green locked by τ and Blue locks on all the data items Blue or Yellow locked by τ. Then all transactions in Before(T) are given White locks on the readset of T, Blue locks on the writeset of T. Finally, all transactions in Before(T) get White locks on all data items White locked by T, and
Blue locks on all data items Blue locked by T. Note that during this last two steps of the lock acquisition phase of transaction T, some transactions other than T get some Blue and White locks. (These other transactions are the transactions in Before(T).) After lock inheritance, the transaction actually starts executing, entering its processing phase.

The algorithms stated above are formally restated in pseudo Pascal. Some of the abbreviations used are:

- WL $\rightarrow$ White Lock
- WL-ed $\rightarrow$ White Locked
- WLS(T) $\rightarrow$ Set of data items White locked by T
- LOCK(White, x) $\rightarrow$ Obtain a White lock on data item x. If lock unavailable due to a conflict, wait until it can be obtained.

(similar, for all other colors)

i) Getting Yellow Locks:

$\text{Before}(T) \leftarrow \emptyset$;
for all $x \in \text{Wr}(T)$ do
begin
  LOCK (Yellow, x);
  $\text{Before}(T) \leftarrow \text{Before}(T) \cup \{ T_i \mid x \text{ is WL-ed or BL-ed by } T_i \}$
end;

ii) Getting Green Locks:

$\text{After}(T) \leftarrow \emptyset$;
for all $x \in (\text{Rd}(T) - \text{Wr}(T))$ do
begin
  LOCK (Green, x);
  $\text{Before}(T) \leftarrow \text{Before}(T) \cup \{ T_i \mid x \text{ is BL-ed by } T_i \}$;
  After(T) $\leftarrow$ After(T) $\cup \{ T_i \mid x \text{ is YL-ed by } T_i \}$
end;
iii) Validation and lock inheritance:

\[
\begin{align*}
\text{if } (\text{After}(T) \cap \text{Before}(T) \neq \emptyset) & \\
& (* \text{ Validation } *) \\
\text{then RESCHEDULE } T & \\
\text{else} & \\
& \begin{align*}
& \text{begin} \\
& \text{for all } \tau \in \text{After}(T) \text{ do} \\
& \quad \begin{align*}
& \text{begin} \\
& \text{Wait for } \tau \text{ to reach locked point;} \\
& \text{BLS}(T) \leftarrow \text{BLS}(T) \cup \text{BLS}(\tau) \cup \text{Wr}(\tau); \\
& (* \tau \text{ gets Blue locks on the writeset of } \tau \\
& \text{ and all items Blue locked by } \tau *) \\
& \text{WLS}(T) \leftarrow \text{WLS}(T) \cup \text{WLS}(\tau) \cup \text{Rd}(\tau) \\
& (* \tau \text{ gets White locks on the readset of } \tau \\
& \text{ and all items White locked by } \tau *) \\
& \text{end;}
\end{align*}
& \text{end;}
& \text{end;}
& \text{end;}
& \text{end;}
\end{align*}
\]

\[
\begin{align*}
& \text{for all } \tau \in \text{Before}(T) \text{ do} \\
& \text{if } \tau \text{ has not terminated then} \\
& \text{begin} \\
& \text{BLS}(\tau) \leftarrow \text{BLS}(\tau) \cup \text{BLS}(T) \cup \text{Wr}(T); \\
& (* \tau \text{ gets Blue locks on all items } \\
& \text{ Blue locked by } T \text{ and the writeset of } T *) \\
& \text{WLS}(\tau) \leftarrow \text{WLS}(\tau) \cup \text{WLS}(T) \cup \text{Rd}(T) \\
& (* \tau \text{ gets White locks on all items } \\
& \text{ White locked by } T \text{ and the readset of } T *) \\
& \text{end}
& \text{end;}
\end{align*}
\]

We stress that there is no assumption of atomicity of any part of the above Transaction Manager algorithms, except in the test and set needed for the implementation of the LOCK function. The LOCK function is the standard locking primitive. If the lock requested cannot be granted due to a conflict, then the process is suspended (or queued) until the lock can be granted. These algorithms can be executed concurrently with all the activities of the other transaction on the database system, including lock acquisition phases of other transactions.

In our protocol, locks can be held only by unterminated transactions. Hence if a transaction \( \tau \) in Before(T) commits before T gets to do lock inheritance, then \( \tau \) does not have to be
given any locks during the lock inheritance of T. Also, during lock inheritance, a transaction T
gets Blue/White locks on items locked by transactions in After(T). But all transactions in
After(T) are guaranteed to be live\(^{\dagger}\) when transaction T is in the lock inheritance phase
(Lemma 3), and thus we do not have to keep records of the locks held by dead transactions.

2.5. Intuitive Discussion

The transaction manager of the Five Color protocol handles all the locking and con-
currency control needed to run transactions. The transactions themselves do not have any
knowledge of the protocols. A transaction has the following structure:

```
Begin-Transaction ( Readset, Writeset)
...
{ Statements }
...
End-Transaction.
```

The Begin-Transaction statement starts up the lock acquisition phase of the transaction.
The transaction manager does the acquiring of locks (using the readset and writset information
provided as parameters) and then performs the validation and inheritance phases. After all the
preprocessing is completed, the transaction starts executing the statements, needing no further
assistance from the transaction manager. When the End-Transaction statement is reached, the
transaction manager is called upon to do the Yellow to Red lock upgrades, actual writes and
commit.

Though acquiring locks is done on behalf of the transaction, by the transaction manager,
in the rest of our discussions we will refer to this event as “a transaction obtains a lock”
because of simplicity and conceptual clarity.

As described in section 2.1, the basic algorithm that the Five Color Protocol uses is as
follows. First, the writset is locked using the Yellow lock. Then the readset is locked with
Green locks. After the readset is read into local storage, the Green locks are downgraded to
White locks.

After the locks are obtained, the validation and lock inheritance processing is done, the
transaction commences execution. After the execution ends, (commit point) the Yellow locks
are upgraded to Red locks, the data written out, and all locks released. The need for validation,
and how it is done is discussed later in this section.

\(^{\dagger}\)In fact, Lemma 3 shows that the transactions in After(T) will be "active" when T reaches locked point,
which is a stronger condition than "live" during lock inheritance.
Suppose Transaction T₁ is running, and it has released all the Green locks on the readset. Now T₂ can update an item which was read by T₁, making T₂ a logically later transaction in the serialization order. In two-phase locking T₂ has to wait until T₁ releases the read lock, and this may mean waiting until T₁ commits. If there are many reading transactions on database systems, transactions that write can be held up for long periods of time.

In addition, write locks held by transactions updating data items, cause delays for reading transactions. The Yellow locking scheme allows read-only transactions to read data items even in the presence of update transactions. Upgrading of locks are allowed in two-phase protocols but this often leads to trivial deadlocks [GrHoKoOb81, Ko83].

The early release of read locks may allow non-serializable executions, and this has to be avoided. Validation is used to avoid possible inconsistencies. The following example shows the need for validation.

Suppose T₁ is running and has read x and released the Green lock on x. T₁ has a Yellow lock on y which it intends to update later. Now a new transaction T₂ can choose to update x (get a Yellow on x), making T₂ a logically later transaction than T₁. T₂ can also attempt to read (or get a Green lock) on y, an item that is Yellow locked by T₁ (making T₂ come logically before T₁). This would lead to a non-serializable condition.

Early release of read locks and late acquisition of write locks as is done by the Five Color protocol leads to non-serializable executions, a simple case of which has been depicted above. The validation and lock inheritance algorithms have been designed to detect such situations and provide serializability. The algorithms use the following strategy.

A transaction T₁ holds White locks on all items it has read. T₁ also holds White locks on data items read by other concurrent transactions that should come after T₁ in the serialization order. Similarly, T₁ holds Blue locks on all items written (or to be written) by concurrent transactions that should come after T₁ in the serialization order. (This property is proved later in Lemma 4). This implies T₁ “knows” about all the data items read or written by “later” transactions.

Suppose a new transaction T₂ arrives in this situation, and Yellow locks an item that is Blue or White locked by T₁. This implies T₂ will update an item that has been read or written by some transaction that is after T₁ in the serialization order. Thus T₂ should be after T₁ in the serialization order. As expected, the Yellow locking of this item causes T₁ to be placed in Before(T₂) (because T₁ is before T₂ in the serialization order). In fact all transactions such as T₁ which should come before T₂ get into Before(T₂).

The fact that T₁ holds Blue locks on data items that have not yet been updated can lead to an anomaly in certain situations. See section 4 for a modification that avoids this.
Now \( T_2 \) can cause a non-serializable condition by attempting to \textcolor{Green}{green} lock an item, which is already \textcolor{Yellow}{yellow} locked by a transaction such as \( T_1 \). \textcolor{Green}{Green} locking a \textcolor{Yellow}{Yellow} locked item implies that \( T_2 \) is getting a view of the database as it was before \( T_1 \) updates the item it \textcolor{Yellow}{yellow} locked, thus \( T_2 \) should precede \( T_1 \). But this action would cause \( T_1 \) to get into \( \text{After}(T_2) \), and \( T_2 \) would fail validation.

The fact that each transaction \( T_1 \) has \textcolor{White}{white} and \textcolor{Blue}{blue} locks on the read and write sets of all transactions that come after \( T_1 \) is ensured as follows: Whenever a new transaction, \( T_2 \), acquires the \textcolor{Green}{green} and \textcolor{Yellow}{yellow} locks, its \( \text{After}(T_2) \) set contains all concurrent transactions that should come after \( T_2 \) in the serialization order. \( T_2 \) then acquires \textcolor{White}{white} (and \textcolor{Blue}{blue}) locks on the readset (and writerset) of all transactions in \( \text{After}(T_2) \). Similarly, it gives all the transactions in \( \text{Before}(T_2) \) \textcolor{White}{white} locks on all its \textcolor{White}{white} locked objects, and \textcolor{Blue}{blue} locks on all its \textcolor{Blue}{blue} locked objects and \( \text{Wr}(T_2) \).

The maintenance of the \textcolor{White}{white} and \textcolor{Blue}{blue} locks might seem contrived and unnecessary, but it has to be done to prevent conditions typified by the following example. Consider three transactions, \( T_1, T_2, T_3 \) having the following traces:

\[
\begin{align*}
T_1 &: R_1(X) \ W_1(Y) \\
T_2 &: W_2(X) \ W_2(Z) \\
T_3 &: R_3(Y) \ W_3(Z)
\end{align*}
\]

A particular sequence of execution under the \textit{Five Color} protocol is depicted in Figure 4.

When \( T_3 \) arrives \( T_2 \) has terminated and \( T_1 \) is executing. Since \( T_2 \) has updated \( X \), and \( T_1 \) has read an older value, \( T_2 \) must be after \( T_1 \) in the serialization order. The locking rules will allow \( T_3 \) to read \( Y \) as it existed before \( T_1 \) updates it, and update \( Z \) which has already been updated by \( T_2 \). However, this is non-serializable. Since at this point \( T_2 \) is no longer alive, and \( T_1 \) does not touch \( Z \), there seems to be no basis for disallowing \( T_3 \) from reading \( Y \) and writing \( Z \), unless the \textcolor{Blue}{blue} and \textcolor{White}{white} locks, and the \text{Before} and \text{After} sets are used.

When \( T_3 \) \textcolor{Yellow}{yellow} locked \( Z \), \( Z \) was \textcolor{Blue}{blue} locked by \( T_1 \). This caused \( T_1 \) to become a member of \( \text{Before}(T_3) \). When \( T_3 \) \textcolor{Green}{green} locked \( Y \), since \( Y \) was \textcolor{Yellow}{yellow} locked by \( T_1 \), \( T_1 \) became a member of \( \text{After}(T_3) \). Now that the intersection of \( \text{Before}(T_3) \) and \( \text{After}(T_3) \) is not empty, \( T_3 \) fails validation, and the non-serializable execution is not allowed. This is an instance of \textit{triggering} that has been mentioned in section 2.2 and discussed further in section 7.

3. Properties and Correctness

We now state the properties of the \textit{Five Color} protocol and prove it achieves serializability. The following Lemmas are useful to understand the details of the algorithms, but the actual proofs may be skipped at the first reading. In order to show that the \textit{Five Color} protocol assures serializability, we define a standard precedence relation \( \rightarrow \) based on the well known
<table>
<thead>
<tr>
<th>T&lt;sub&gt;1&lt;/sub&gt;</th>
<th>T&lt;sub&gt;2&lt;/sub&gt;</th>
<th>T&lt;sub&gt;3&lt;/sub&gt;</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Arrives, *Yellow* locks Y  
*Green* locks X  
reads X  
Converts *Green* on X to *White*. | Arrives, *Yellow* locks X  
*Yellow* locks Z  
Updates X  
Updates Z  
Terminates | T<sub>1</sub> ∈ Before(T<sub>2</sub>) |                                                                                           |
| Gets *Blue* lock on X and Z                                              |                                                                             |                                                                             | T<sub>1</sub> ∈ After(T<sub>3</sub>)  
(fails validation,  
see discussion.) |
| Executing ....                                                           | Arrives  
*Yellow* locks Z  
*Green* locks Y  
..... |                                                                             |                             |

Figure 4: Execution of the example history under the *Five Color* protocol.

notions of read-write and write-write conflicts [EsGrLoTr76, BeGo80]. The acyclicity of this relation implies serializability (though not vice versa [Pa79]). We will show that the *Five Color* protocol assures acyclicity of the → relation.

All transactions that partake in the relations in the definitions are assumed to be transactions that have already passed the validation phase. Transactions that have not passed validation also generate relationships with other transactions, but since these transactions do not have any effect on the database, we choose to ignore them in the correctness proofs.
Definitions:

- A transaction is active if it has reached locked point, but has not completed acquiring Red locks.
- Define a binary relation \( (\rightarrow) \) over transactions. Two transactions \( T_i \) and \( T_j \) are related by the precedence relation \( (T_i \rightarrow T_j) \), if and only if:
  i) \( T_i \) reads some data item \( x \) from the database, and at some later point \( T_j \) writes \( x \) into the database (r-w conflict), or
  ii) \( T_i \) writes some data item \( x \) into the database, and at some later point \( T_j \) reads \( x \) from the database (w-r conflict), or
  iii) \( T_i \) writes some data item \( x \) into the database, and at some later point \( T_j \) writes \( x \) into the database (w-w conflict).

The graph defined by the precedence relation is called the serializability graph [BeHaGo87].

Lemma 1

L.1: The relation \( T_i \rightarrow T_j \) occurs if and only if one of the following cases take place:

i) \( T_i \) gets a Green lock on \( x \) and then \( T_j \) gets a Yellow lock on \( x \) after \( T_i \) releases the Green lock. This arc is defined as of type \( [\alpha \rightarrow \beta | \alpha G, \beta Y] \).

ii) \( T_j \) gets a Yellow lock on \( x \), then while \( T_j \) holds the Yellow lock, \( T_i \) gets a Green lock on \( x \), and then \( T_j \) converts the Yellow lock into a Red. This arc is defined as of type \( [\alpha \rightarrow \beta | \beta Y, \alpha G, \beta R] \).

iii) \( T_i \) gets a Yellow lock on \( x \), and later, after \( T_i \) unlocks \( x \), \( T_j \) gets a Green lock on \( x \). This arc is defined as of type \( [\alpha \rightarrow \beta | \alpha Y, \alpha R, \beta G] \).

iv) \( T_i \) Yellow locks \( x \), and later \( T_j \) Yellow locks \( x \). This arc is defined as of type by \( [\alpha \rightarrow \beta | \alpha Y, \beta Y] \).

Proof of L.1:

Simple, but lengthy.

The two cases ii) and iii) (above) look similar but cause very different results. In ii), \( T_j \) gets a Yellow lock on \( x \) and then, while the Yellow lock is held, \( T_i \) gets a Green lock on \( x \). In

\[\text{The notations such as } [\alpha \rightarrow \beta | \alpha G, \beta Y] \text{ denotes types of edges. If two transactions } T_1 \text{ and } T_2 \text{ are related as } T_1 \rightarrow T_2, \text{ this relation can be caused in many ways. } \alpha \text{ stands for the transaction to the left of the } \rightarrow \text{ symbol, and } \beta \text{ is the transaction on its right. } R, \text{ Y and } G \text{ denote getting a Red, Yellow or Green lock. The sequence to the right of the '}' shows the sequence of lock acquisitions by } \alpha \text{ and } \beta \text{ which led to the formation of the } \rightarrow \text{ relation.} \]
this case, \( T_j \) will update \( x \), but \( T_i \) gets to see \( x \) as it existed before \( T_i \) updated it. Hence the serialization order of the transactions should be \( T_i \rightarrow T_j \). However in case iii), under similar circumstances, if \( T_j \) gets a Yellow lock, and after this Yellow lock has been converted to Red, and released, \( T_i \) gets a Green lock on \( x \), then the serialization order should obviously be \( T_j \rightarrow T_i \). (The roles of \( T_i \) and \( T_j \) in the second example has been reversed, to be similar to the first example.)

The arcs of the serializability graph \((\rightarrow)\) are caused by r-w, w-r and w-w conflicts. These conflicts happen when both the transactions have actually processed the conflicting reads and writes. However for ease of modeling we will assume that the arcs are created earlier. We define that an arc \( T_i \rightarrow T_j \) is created when either \( T_i \) or \( T_j \) reaches locked point, whichever is later. (\( T_i \) and \( T_j \) must of course conflict as stated in Lemma 1.) This occurs after all the locks for \( T_i \) and \( T_j \) have been obtained, but before any actual conflict due to actual reading or writing has taken place. Also, if a transaction has not reached locked point, there is no arc either from or to it.

Note that the serializability graph thus formed is not identical to the graph formed by the r-w, w-r and w-w conflicts at some given point in time. This is because the arcs of the precedence graph are created prior to occurrence of the conflicts. Thus this serializability graph is in fact a superset of the conflict graph, which will become identical to the conflict graph when all activity on the database ceases. However our correctness criterion is the acyclicity of the conflict graph (see below) and acyclicity of the serializability graph implies acyclicity of the conflict graph.

**Lemma 2**

L.2: If \( T_i \rightarrow T_j \) and \( T_j \) reached its locked point before \( T_i \) did, then the arc can only be of type

\[ [\alpha \rightarrow \beta | \beta Y, \alpha G, \beta R] \]

**Proof of L.2:**

Since \( T_j \) has reached its locked point, it has obtained all its locks. The arc \( T_i \rightarrow T_j \) could have been caused by four cases (Lemma 1). Consider each case separately:

\[ [\alpha \rightarrow \beta | \alpha G, \beta Y] \]

After \( T_j \) gets a Green lock on \( x \), \( T_j \) cannot get a Yellow lock on \( x \) until \( T_i \) converts the Green lock to a White lock. Hence \( T_j \) cannot reach the locked point before \( T_i \) does and this case is impossible.

\[ [\alpha \rightarrow \beta | \beta Y, \alpha G, \beta R] \]

This case is possible, as \( T_i \) may obtain a Green lock on \( x \) after \( T_j \) has placed a Yellow
lock on x.

\[ \alpha \to \beta \mid \alpha Y, \alpha R, \beta G \]

In this case \( T_1 \) has to get a Red lock on x before \( T_j \) gets a Green lock on x. This makes it impossible for \( T_j \) to get to the locked point before \( T_1 \), thus this case is impossible.

\[ \alpha \to \beta \mid \alpha Y, \beta Y \]

Again, \( T_1 \) has to release the Yellow lock on x before \( T_j \) can place another Yellow lock on x. Hence \( T_j \) cannot reach its locked point before \( T_1 \), making this case impossible too.

Thus only the second case can take place, and hence the arc is of type \[ \alpha \to \beta \mid \beta Y, \alpha G, \beta R \].

Lemma 3

L.3: If \( T_i \to T_j \) and \( T_j \) reached its locked point before \( T_i \) did, then \( T_j \) is still active when \( T_i \) reaches its locked point.

Proof of L.3

The arc \( T_i \to T_j \) is of the type \[ \alpha \to \beta \mid \beta Y, \alpha G, \beta R \] (Lemma 2). Thus \( T_j \) gets a Yellow lock first, then \( T_i \) gets a Green lock and finally \( T_j \) upgrades its Yellow lock to Red. After \( T_i \) gets a Green lock on x, \( T_j \) cannot convert its Yellow lock to a Red lock until \( T_i \) converts its Green lock to a White lock. Thus when \( T_i \) reaches its locked point, \( T_j \) must be active.

Lemmas 2 and 3 show that unlike two-phase locking the serializability graph can grow "backwards" (see section 5). A transaction \( T_2 \), which arrives later than transaction \( T_1 \), may precede \( T_1 \) in the precedence order, if \( T_1 \) is active when \( T_2 \) arrives.

Lemma 4

L.4: If \( T_1 \to T_2 \to \ldots \to T_n \) is a path in the serializability graph, and \( T_1 \) is active, then

L.4.1: \( T_1 \) possesses White locks on \( \text{Rd}(T_n) \) (i.e. \( \text{Rd}(T_n) \subseteq \text{WLS}(T_1) \)), and
L.4.2: \( T_1 \) possesses Blue locks on \( \text{Wr}(T_n) \) (i.e. \( \text{Wr}(T_n) \subseteq \text{BLS}(T_1) \)).

Proof of L.4.1:

Proof is by induction on \( N \), the number of transactions in the path. We show it is true for \( n = 2 \), then assume it is true for \( n = N \) and show it follows for \( n = N + 1 \).
\[ n = 2 \]

Let \( T_1 \rightarrow T_2 \), where \( T_1 \) is active. The arc in the path can be of four types. Consider each case separately:

\[ \alpha \rightarrow \beta \mid \alpha G, \beta Y \]

\( T_2 \) can obtain the Yellow lock on the data item (say \( x \)) after \( T_1 \) has converted the Green lock it was holding to a White lock. When \( T_2 \) gets a Yellow lock on \( x \) while \( T_1 \) holds a White lock on it, \( T_1 \) is added to Before\( (T_2) \). Then \( T_1 \) inherits White locks on \( T_2 \)'s read-set. Hence \( \text{Rd}(T_2) \subseteq \text{WLS}(T_1) \).

\[ \alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R \]

When \( T_1 \) gets a Green lock on \( x \) while \( T_2 \) is holding a Yellow lock on \( x \), \( T_2 \) gets into the set After\( (T_1) \). Then \( T_1 \) inherits White locks on the readset of \( T_2 \). Hence \( \text{Rd}(T_2) \subseteq \text{WLS}(T_1) \).

\[ \alpha \rightarrow \beta \mid \alpha Y, \alpha R, \beta G \]

\( T_1 \) cannot be active in this case, as \( T_1 \) has released a Red lock before the edge could be created and hence has passed into inactive state.

\[ \alpha \rightarrow \beta \mid \alpha Y, \beta Y \]

\( T_1 \) cannot be active in this case as \( T_1 \) has released the Yellow lock, which implies \( T_1 \) has upgraded the Yellow to Red and released the Red lock.

Thus \( T_1 \) has White locks on the readset of \( T_2 \).

\[ n = N \]

We assume Lemma 4 holds for some value \( N \) of \( n \).

\[ n = N + 1 \]

Consider a path consisting of \( n (=N+1) \) transactions. By definition all the transactions in this path have reached their locked points. Let \( T_k \) be the transaction that reached its locked point last, amongst all the transactions in the path:

\[ T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_{k-1} \rightarrow T_k \rightarrow T_{k+1} \rightarrow T_{k+2} \rightarrow \ldots \rightarrow T_n \]
Now there are three cases:

1] \(2 < k < n-1\)
2] \(k=1\) or \(2\) or \(n-1\) or \(n\).

First consider case 1):

Consider the instance when \(T_k\) just reached its \textit{locked point}. At this point the arcs:

\[T_{k-1} \rightarrow T_k \quad \text{and} \quad T_k \rightarrow T_{k+1}\]

are created. Thus prior to this there were two shorter paths, each path having a length less than \(N\), for which Lemma 4 holds.

a) \(T_1 \rightarrow \ldots \rightarrow T_{k-1}\) and
b) \(T_{k+1} \rightarrow \ldots \rightarrow T_n\).

For path a), \(T_1\) is alive (by definition) and thus \(T_1\) has \textit{White} locks on \(Rd(T_{k-1})\) and \textit{Blue} locks on \(Wr(T_{k-1})\), since \((k-1) < N\). As \(T_k\) reached \textit{locked point} later than \(T_{k+1}\), it follows that \(T_{k+1}\) was active when \(T_k\) reached \textit{locked point}, and thus had \textit{White} locks on \(Rd(T_n)\), and \textit{Blue} locks on \(Wr(T_n)\) (because of path b).

By Lemma 2, the arc \(T_k \rightarrow T_{k+1}\) must be of type \([\alpha \rightarrow \beta \mid \alpha G, \beta R]\). Hence \(T_k\) gets a \textit{Green} lock on some \(x\) that is \textit{Yellow} locked by \(T_{k+1}\). Thus \(T_{k+1}\) is in \(\text{After}(T_k)\), and since \(T_k\) gets \textit{White} locks on all items \textit{White} locked by \(T_{k+1}\), \(T_k\) gets \textit{White} locks on \(Rd(T_n)\).

The \(T_{k-1} \rightarrow T_k\) arc can be of four types. Treating them separately:

\([\alpha \rightarrow \beta \mid \alpha G, \beta Y]\)

In this case \(T_k\) sets a \textit{Yellow} lock on a data item \(x\), which is in \(Rd(T_{k-1})\), and thus \(x\) is \textit{White} locked by \(T_1\). This makes \(T_1 \in \text{Before}(T_k)\), and thus \(T_1\) inherits \textit{White} locks on all \(WLS(T_k)\), which contains \(Rd(T_n)\). Thus \(T_1\) gets \textit{White} locks on \(Rd(T_n)\).

\([\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]\)

This cannot happen as \(T_{k-1}\) must reach \textit{locked point} before \(T_k\). For if \(T_k\) gets a \textit{Yellow} lock, and then \(T_{k-1}\) gets a \textit{Green} lock, \(T_k\) becomes a member of the set \(\text{After}(T_{k-1})\). Now \(T_{k-1}\) has to wait for \(T_k\) to reach \textit{locked point}, before it can reach \textit{locked point}. (This is a condition during lock inheritance, please see the algorithm description at the end of section 2.3.)
In this case $T_k$ sets a $\text{Green}$ lock on a data item $x$, which is in $\text{Wr}(T_{k-1})$, and thus $x$ is
$\text{Blue}$ locked by $T_1$. This causes $T_1 \in \text{Before}(T_k)$, and thus $T_1$ inherits $\text{White}$ locks on all
$\text{WLS}(T_k)$, which contains $\text{Rd}(T_n)$. Thus $T_1$ gets $\text{White}$ locks on $\text{Rd}(T_n)$.

In this case $T_k$ sets a $\text{Yellow}$ lock on a data item $x$, which is in $\text{Wr}(T_{k-1})$, and thus $x$ is
$\text{Blue}$ locked by $T_1$. This makes $T_1 \in \text{Before}(T_k)$, and thus $T_1$ inherits $\text{White}$ locks on all
$\text{WLS}(T_k)$ which contains $\text{Rd}(T_n)$. Thus $T_1$ gets $\text{White}$ locks on $\text{Rd}(T_n)$.

Thus for all cases, $T_1$ has $\text{White}$ locks on readset of $T_n$.

For the situations where $k = 1$, or 2, or $n-1$, or $n$; the above proof can be modified. We
sketch the case for $k=1$, and leave the rest to the reader.

When $k=1$, we have one path before $T_1$ reaches locked point:

$$T_2 \rightarrow \ldots \rightarrow T_n$$

Since this path has less than $N$ transactions, Lemma 4 holds; and $T_2$ has $\text{White}$ locks on $\text{Rd}(T_n)$. It can then be shown that when $T_1$ reaches locked point ($T_2$ is active) $T_1$ inherits all
the $\text{White}$ locks from $T_2$, making $T_1$ have $\text{White}$ locks on $\text{Rd}(T_n)$.

Proof of L.4.2:

Similar to proof of L.4.1. Substitute $\text{Blue}$ for $\text{White}$ and writeset for readset in proof of L.4.1
above.

Lemma 4 shows the most important property of the $\text{Five Color}$ protocol. This implies that
if a transaction $T_1$ is active, it "knows" about the read and write set of all transactions that
come after $T_1$. This property is used to achieve serializability by causing a validation conflict
when a cycle is created by some transaction.

Theorem 1:

The protocol ensures serializability.

Proof

We show that the serializability graph is acyclic. The proof is by contradiction. Assume
there can be a cycle in the $\rightarrow$ relation. Choose a minimal cycle:
T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_{k-1} \rightarrow T_k \rightarrow T_1

Assume that T_k is the transaction to reach its locked point last, compared to all the other transactions participating in the cycle. It will be shown that if T_k accessed data items in a manner that caused the cycle in the \rightarrow relation, then T_k would have been rescheduled at the validation phase.

T_k causes the creation of two arcs, which cause the cycle. They are the arcs:

A1: T_{k-1} \rightarrow T_k$

A2: T_k \rightarrow T_1$

As T_k is the last transaction to reach its locked point, T_1 must have reached its locked point before T_k, hence (by Lemma 2) the arc \langle T_k \rightarrow T_1 \rangle must be of type \([\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]\).

Thus T_k gets a Green lock on a data item Yellow locked by T_1. Hence T_1 \in After(T_k).

Also, by Lemma 3, T_1 must have been active when T_k reached its locked point. Thus by Lemma 4, T_1 has White locks on Rd(T_{k-1}) and Blue locks on Wr(T_{k-1}).

The arc \langle T_{k-1} \rightarrow T_k \rangle could be of four types. Let us treat them separately:

\([\alpha \rightarrow \beta \mid \alpha G, \beta Y]\)

In this case T_1 \in Before(T_k) (see proof of Lemma 4). Thus T_1 \in (Before(T_k) \cap After(T_k)), and hence T_k should have been rescheduled at validation, and the cycle could not have resulted.

\([\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]\)

This type of arc could have been caused only if T_k reached locked point before T_{k-1}, which is not the case.

\([\alpha \rightarrow \beta \mid \alpha Y, \alpha R, \beta G]\)

In this case T_1 \in Before(T_k) (see proof of Lemma 4). Thus again T_1 \in (After(T_k) \cap Before(T_k)) and T_k should have been rescheduled.

\([\alpha \rightarrow \beta \mid \alpha Y, \beta Y]\)

In this case, again T_1 \in Before(T_k). Rest as above.

Thus there can be no cycle in the \rightarrow relation under the protocol, and hence the protocol ensures consistency of updates.

4. A Modification

The algorithm as described has an undesirable feature. It does not hamper consistency but may cause more aborts than necessary. Consider the following situation:

1) T_1 gets Green lock on x
2) T_1 downgrades Green lock to White lock
3) T_2 gets Yellow lock on x
4) \( T_3 \) gets *Green* lock on \( x \)

At step 3, \( T_1 \) is in Before(\( T_2 \)) and thus gets a *Blue* lock on \( x \). When \( T_3 \) gets a *Green* lock on \( x \), Before(\( T_3 \)) contains \( T_1 \). Actually \( T_1 \) and \( T_3 \) are unrelated. The anomaly exists as \( T_2 \) has *not yet written* \( x \) when \( T_3 \) reads \( x \). There would have been no problem if \( T_2 \) had terminated before \( T_3 \) read \( x \), and in this case \( T_3 \) would have to be after \( T_1 \).

Thus in a path of transaction \( T_1 \rightarrow T_2, ..., \rightarrow T_k \), \( T_1 \) should have *Blue* locks only on those data items that have been updated by \( T_2, ..., T_k \), and not on all data items in the writeset of \( T_2, ..., T_k \).

The following is a very brief description of a method to prevent the above anomaly. Introduce another type of lock, called a *I-Blue* lock (or Intent-*Blue* lock). During lock inheritance, *Blue* locks are obtained on items already *Blue* locked by other transaction, while *I-Blue* locks are obtained on the writesets (uncommitted updates) of the transactions concerned. When a transaction commits, all the *I-Blue* locks held on its writeset by other transactions are changed to *Blue* locks. All other algorithms remain the same.

This modification is not incorporated in the algorithm as described in section 2. This is to avoid introduction of extra complexity, which has no bearing on the correctness of the protocol, and simplify understanding of the protocol. This is not a correctness issue, nor an important point in the concepts used in the *Five Color* protocol.

5. Deadlocks

In the *Five Color* protocol there is potential for two types of deadlocks, one due to locking and the other due to waiting for other processes to reach their locked points. Let us address them separately.

The first form of deadlock is the one encountered in traditional locking protocols, and is caused by transactions in a circular wait, trying to obtain locks. This form of deadlock can be prevented in this protocol, using the predeclared read and write sets. We define an ordering of resources, and acquire all locks in an increasing order of resources. First all the *Yellow* locks are obtained, in increasing order, and then all the *Green* locks are obtained in that order. The *Yellow* to *Red* conversion is also done in the increasing order of resources (data items).‡

**Theorem 2**

The pre-ordered locking strategy used in the *Five Color* protocol is deadlock free.

‡ If we use only one type of (exclusive) lock, this strategy of obtaining locks in a predefined fashion is known to be deadlock free. However it is not true in general, where several types of locks with various compatibilities are used. In our case, however, it is deadlock free, as shown in theorem 2.
**Definition**

Suppose transaction $T_i$ requests a lock on item $x$ in mode $m_1$, and item $x$ is already locked in mode $m_2$ by transaction $T_j$. If mode $m_1$ is incompatible with mode $m_2$ we say $T_i$ waits-for $T_j$. The waits-for relation is denoted as $T_i \rightarrow T_j$. The directed graph defined by the waits-for relation over all transactions is called the waits-for graph.

**Proof**

It can be shown that a cycle in the waits-for graph is an instance of a deadlock condition. Suppose deadlocks can take place in our system. Consider an instance of a deadlock involving $k$ transactions, with the cycle in the waits-for graph being:

\[ T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \rightarrow T_1 \]

**Step i)**

Suppose none of the transactions $T_1$ to $T_k$ are in the process of acquiring Red locks, that is they are in the Green or Yellow locking phase. Now $T_1$ to $T_k$ cannot be all in the Green lock or Yellow lock acquiring phase. (If there is one type of locking, this strategy is known to be deadlock free). Thus some transactions are waiting for a Green lock, and others are waiting for a Yellow lock. Hence at some point, a transaction $T_i$ is waiting for a Green lock on a data item $x$, for which $T_j$ is holding a Yellow lock. This is a contradiction as the Green lock is compatible with an existing Yellow lock. Thus at least one transaction must be holding a Red lock.

**Step ii)**

Suppose one of the transactions, say $T_1$, is in the process of upgrading its Yellow locks to Red locks, and holds at least one Red lock. A transaction which is trying to upgrade a Yellow lock on $x$, to a Red lock will have to wait only if some other transaction holds a Green lock on $x$ (as no other transaction can hold a Yellow lock on $x$). Thus if $T_1$ is waiting for $T_2$, then $T_2$ must be holding a Green lock. Since $T_2$ is also waiting, it must be waiting to get another Green lock, since acquiring of Green locks are done after acquiring of Yellow locks, and all Green locks are released before any Red locks are acquired. Thus $T_2$ is in the phase of acquiring Green locks.

Similarly, a transaction trying to acquire a Green lock on $x$ will wait for another transaction only if that transaction holds a Red lock on $x$, as Red is the only lock incompatible with the Green lock.

Thus if $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \rightarrow T_1$ is a cycle of waiting transactions and $T_1$ is in the Red lock acquiring phase, then so is $T_3$, $T_5$, ..., and $T_2$, $T_4$, ..., are in the Green lock acquiring phase (that is, the cycle comprises only of transactions in the Green or Red lock acquiring
phases).

The proof that there cannot be a cycle in a set of transactions having the above properties, is very similar to the proof that there cannot be a cycle in a set of transactions acquiring exclusive locks in a predefined order, and is not included here for brevity [Ha68].

The second form of deadlock is peculiar to this protocol. This form of deadlock involves one or more transactions in the lock inheritance phase. Note that in the lock inheritance phase, a transaction T has to wait for all transactions in After(T) to reach locked point. This can cause deadlocks. The following is an example of a deadlock caused by two transactions in the lock inheritance phase.

i) $T_1$ *Yellow* locks x

ii) $T_2$ *Yellow* locks y

iii) $T_1$ *Green* locks y

iv) $T_2$ *Green* locks x

In this sequence of events the following problem takes place. Steps i) and ii) cause no surprises, but in Step iii) as $T_1$ tries to *Green* lock y, which is *Yellow* locked by $T_2$, $T_2$ becomes a member of After($T_1$). Similarly, when $T_2$ *Green* locks x, $T_1$ becomes a member of After($T_2$).

Now both transactions will pass the validation, and wait for all transactions in their After sets to reach locked point. $T_1$ will thus wait for $T_2$ to reach locked point before it can reach locked point, and vice versa. Thus we have a deadlock. This deadlock effectively prevents the non-serializable log that could result if the transactions were allowed to continue.

The deadlock can involve transactions waiting for locks as well as transactions waiting for other transactions to reach locked point. However as proved above there are no deadlocks involving only transactions waiting for locks.

The deadlocks can be detected by standard deadlock detection algorithms, or by timeouts. As the deadlock occurs before the transactions start execution, there is no rollback involved and the overhead suffered is small.

Since the transactions which participate in deadlocks do not do any processing, we believe timeouts may be an easier and more efficient method to deal with deadlocks. Lack of processing means there are no processing delays and the timeouts can be fine tuned better to take into account the locking delays and cause deadlock warnings if something takes too long to happen.

Every deadlock cycle has at least one transaction waiting for the completion of the lock acquisition phase of another transaction. We propose that this is the point where a timeout should be introduced. The delay for the timeout can be a function of the number of locks the
other transaction has to acquire. This will keep the probability of detection of false deadlocks to be low.

6. Livelocks

There is a small chance of livelocks or starvation in this protocol. This is due to the rescheduling of transactions because of validation failure. There is no guarantee that an aborted transaction will finally be able to run. However we feel that the chances of starvation is extremely small. But low probability of starvation is still not a guarantee against starvation, so we propose the following method of avoiding livelocks.

If a transaction gets aborted due to validation failure a large number of times, we label the transaction as *starvation prone*. The writeset of a starvation prone transaction is then expanded to contain its readset. The transaction thus acquires only *Yellow* locks during lock acquisition, and as a result has an empty After set. This transaction is guaranteed to pass validation, avoiding the livelock problem. Also it never waits for any other transaction to complete lock inheritance. It may still, however participate in deadlocks. But note that to detect the deadlocks we are timing out and aborting a transaction that waits for transactions in its After set. Since the starvation prone transaction has an empty After set it will not get aborted even if it participates in a deadlock. Thus it is guaranteed to run to completion.

7. Discussion

As stated earlier, The *Five Color* protocol differs significantly from the two-phase locking protocol in the way the serializability graph may grow. In the two-phase locking protocol the precedence arcs are created when a transaction locks a data item. When a transaction locks an item (or upgrades a lock) it places itself after some other transaction, never before. Thus we say the serializability graph grows only in the *forward* direction. In fact this is the property of the two-phase locking protocol that ensures serializability.

Thus, in our protocol, a path under the precedence order can grow in *both* directions. Suppose transaction T has reached its locked point and possesses a *Yellow* lock on x. A new transaction τ arrives and gets a *Green* lock on x. When τ reaches locked point, the arc τ → T is created. Now, even after T terminates, as long as τ is active, τ' may come and place itself before τ, and hence before T. In this way, it is possible for future transactions to be logically placed in the past.

Some locking protocols for example the Bayer et. al. protocol and the 2V2PL protocol discussed in section 1.4 allow one transaction to read an older value (before value) of a data item that is being updated by another transaction. This situation is the same as setting a *Green* lock on a *Yellow* locked item. However there are some important differences.
Since the Bayer and 2V2PL protocols are two-phase, they do not allow the serializability graph to grow in both directions. A transaction $T_1$ may read the before value of $X$ while $T_2$ is updating $X$, but in this case $T_2$ must wait for $T_1$ to commit before it can commit. The Five Color protocol does not place this restriction.

Multiversion timestamp protocols and the Sterns et. al. protocols also allow transactions to read before values. It is difficult to compare these protocols with the Five Color protocol as the mechanisms used by them are quite different.

Thus the basic mechanism by which two-phase locking prevents serializability is not present in our protocol. Serializability is ensured, in this case by the Blue and White locks, and the validation procedure. Intuitively, if $T_1 \rightarrow \ldots \rightarrow T_2$ is a chain of transactions, then $T_1$ "knows" about $Rd(T_2)$ and $Wr(T_2)$, because it has White and Blue locks, respectively, on these data items. If any transaction $\tau$ attempts to read any data item in $Wr(T_2)$ (or write any item in $Rd(T_1)$) then due to the "triggering" caused by Green (Yellow) locking of a Blue (White) locked item, $T_1$ becomes a member of Before($\tau$) and $T_1$ inherits White (Blue) locks on $Rd(\tau)$ ($Wr(\tau)$). Thus information about the read and write sets flow up a chain in the form of "inherited" White and Blue locks. Now if $\tau$ may cause a cycle in the $\rightarrow$ relation by attempting to read an item Yellow locked by $T$, then $T$ would become a member of After($\tau$) and violate the validation constraint. The other cases of information flow when the chain grows in the reverse direction, or when two chain as concatenated by a transaction is similar and is covered in the proofs (Lemma 4).

This property of the Five Color protocol allows the protocol to produce non-two-phase histories. For example, the Five Color protocol allows non-strictly-serializable\(^\dagger\) histories [BeGo80]. The following is an example of a non-strictly-serializable history that is allowed by the Five Color protocol. Some timestamp based protocols allow the following log, if the timestamp ordering is $T_3 < T_2 < T_1$.

<table>
<thead>
<tr>
<th>Log</th>
<th>Serial order</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1(x)$ $R_2(y)$ $W_1(y)$ $R_3(z)$ $W_2(z)$</td>
<td>$T_3 \rightarrow T_2 \rightarrow T_1$</td>
</tr>
</tbody>
</table>

Note: Though $T_1$ completes execution before $T_3$ commences, $T_3$ precedes $T_1$ in the serialization order.

\(^\dagger\) A log $L$ is strictly-serializable if there exists a serial order $L_s$ of $L$ such that if $T_1$ and $T_2$ are in $L$ and their action do not interleave, then $T_1$ and $T_2$ appear in the same order in $L_s$ as in $L$. A log that is serializable, but does not have the above property is non-strictly-serializable.
properties of the *Five Color* protocol. Some of these properties may lead to a higher concurrency level for the *Five Color* protocol, than what is possible for the family of two-phase protocols. Since the sets of histories produced by the *Five Color* protocol and the two-phase locking protocols are incomparable (that is, there exist histories produced by two-phase locking that are not produced by *Five Color* and vice versa), we cannot conclusively argue superiority of one over the other.

- Early release of read locks.
  The *Five Color* protocol releases read locks as soon as the data is read. This allows other transactions to update these data items without having to wait for the reading transaction to commit. [In practical two-phase protocols, such as 2V2PL, all locks are held till commit point].

- Allowing reading of data items to be written later.
  This allows reading transactions faster access to data items that would have been exclusively locked for quite some time otherwise. The Bayer protocol, Sterns protocol, 2V2PL and some other protocols have this feature. However, in all locking protocols, the updating transaction, then has to wait for the reading transaction to commit before it can commit.

- Non-compatibility of Yellow locks.
  This property prevents some deadlocks. Consider the 2V2PL protocol. If two protocols issue read requests on X and then issues write requests on X, they will both try to upgrade the read locks they posses into write locks, causing a deadlock. This is termed a trivial deadlock [Ko83]. The deadlock is trivial to detect but just as serious as any deadlock, as one of the transactions have to be aborted. The incompatibility of the Yellow lock, avoids chances of trivial deadlocks and causes delay of an update transaction while another is updating the data item (delay is better than deadlock). A simulation study we conducted showed that trivial deadlocks are the most common form of deadlock in 2V2PL type protocols. However, it must be noted that the Five Color protocol avoids this problem since it has the knowledge of read/write sets, which 2V2PL does not.

- Preordering of locking requests.
  This avoids deadlocks due to locking. The avoidance of such deadlocks as well as trivial deadlocks may cause the *Five Color* protocol to have a lower chance of deadlocks than protocols using un-ordered locking. However, any protocol that has knowledge of the read and write sets is able to do ordered locking. When attempting to do ordered locking, care has to be taken to see that locks are not held longer than necessary. The *Five Color* protocol has been designed with that in mind.
7.1. Conclusions

We have presented a locking protocol that uses an unconventional locking strategy, and knowledge about the read and write sets of the transactions to allow non-two-phase locking on a general database. We show that this protocol ensures serializability and has properties that may allow it to perform better than the two-phase locked protocols.

The Five Color protocol is substantially more complicated than the two-phase locking protocols. In fact the simplicity and elegance of the two-phase locking protocols are their major attractions. Nevertheless, we believe that the Five Color protocol is worth serious consideration.

Thus we conclude that it is possible for the Five Color Protocol to achieve more concurrency by anticipating the absence of conflicts in a large number of cases, due to the information available to the transaction manager about the readsets and writesets of the transaction.

8. Acknowledgements

We would like to acknowledge the anonymous referees for their in depth comments and for pointing out several shortcomings that escaped our notice. We would also like to thank Phil Bernstein and Magdi Morsi for their help and comments.

9. Bibliography


A Probe-Based Monitoring Scheme for an Object-Oriented, Distributed Operating System

Partha Dasgupta

Georgia Tech

Introduction

This paper presents an integration of three topics, namely distributed systems, object-oriented design and system monitoring. System monitoring is a functionality that enables a large system to keep track of the health of individual components, especially if it has multiple instances of each type. This is particularly interesting in distributed systems that implement fault tolerance. Given the ability to detect failing, flaky or failed components (software modules and hardware units) the system has the ability to reconfigure the healthy units, on the fly, to work around the faulty ones.

The basic mechanism we propose, for monitoring, is the usage of probes. Probes are a powerful tool in many environments and has been proposed for deadlock detection, debugging, backup processing and so on. [ChMi82] Though the usage of probe-based algorithms can be generalized for many applications, the usage is very dependent on the exact semantics of the probe implementation and the environment in which they function.

In the following pages we explain the use of probes in a particular distributed system framework. The system under consideration is a prototype of a distributed operating system and programming environment we call Clouds. Clouds is a object-oriented system building concept, that is wholly structured on the object concept. This forms a solid base layer for further research into the programming and environment aspects of object-oriented design.

We first present a short summary of the design criteria, goals and architecture of the Clouds system. This will illustrate the framework of fault tolerant distributed system that we have currently. We then present a design for the probe system that will fit elegantly into the existing design. We show how the probes (in this context) can be effectively used for system monitoring, reconfiguration and fault tolerance. The probe system also has some other payoffs like the easy implementations of interactive debugging support.

2. The Clouds Operating System

The Clouds operating system is a distributed, object-oriented operating system that supports objects, nested actions, reconfiguration, fault tolerance, orphan and deadlock detection and integration (location transparency). [Mo81, Al83, Mc84a, DaLeSp85]

The basic building blocks in Clouds are objects, actions and processes. Processes are carriers of the thread of control, on behalf of actions. The actions are atomic units of activity, consisting of a partial order of invocations of operations defined in objects (see sections 2.2 and 2.3).

2.1. The Clouds Architecture

The architecture of a distributed system can be partitioned into two main areas: the hardware configuration and the operating system structure. The operating system structure is loosely described above, as the set of objects, actions and processes. The hardware organization of Clouds is a set of processors, loosely coupled over a medium-to-high-speed network. The prototype configuration is shown in Fig. 1.
The prototype consists of three VAX/750 computers with 3 Meg memory each, connected by an Ethernet. The Ethernet serves both as a back-end network that links the machines of the distributed system, as well as a front end network that the users use to gain access to the system. The front end network is the gateway to the Clouds system, and the user access is through workstations (diskless SUN's or IBM-PC/AT's). Since the users are not hard wired to any site, in case of failures, users can be virtually transparently floated to another serviceable processor. The desktop computers used by the user run a special interfacing software system which can communicate to Clouds and can intelligently take part in system monitoring and can decide to change hosts if necessary, in cooperation with the surviving part of Clouds. [Mc84a]

The disk drives used for secondary permanent storage are dual ported. This allows reassignment of disk drives in case of processor failure and helps in reconfiguration, and leads to high availability of data.

2.2. Objects

The operating system structure of Clouds is based on the action/object paradigm. All permanent system components in Clouds are objects. Objects form a clean conceptual encapsulation of data and programs. They are useful in providing synchronization and recovery as will be described later. The objects are accessed by processes on behalf of actions.

In a simplistic view, an object is an instance of an abstract data type (cosmetically similar to modules in Modula-2 or classes in Simula). The object encapsulates permanent data and a set of routines that can access (read or update) the data. The only access path to the data contained in the object is through the routines (or operations) defined in the object. To the external world, the object is thus an entity providing a set of entry points. [Jo79]

Object instances in Clouds are permanent, that is they exist after they are created and are visible until they are explicitly deleted (like files). Any update caused on the data in an object by an operation invocation is also permanent. Thus all programs, data and files in the Clouds system are objects. For example, a file object is an object containing the file data, along with the read and write operation defined on the data. Thus each object can be tailor made to the application that requires the use of the object, and also tailored to suit the access methods and other update criteria applicable to the data contained in the object.

Clouds objects are more powerful than just a black box containing some procedures and static data. It also contains a stack (temporary data), heap (permanent dynamically allocated data), and powerful support for concurrency control and recovery.

Objects in Clouds are passive, that is they consist of just a memory image, without any processes/servers associated with them. This allows virtually unlimited object instances with little overhead. Also invoking an object does not involve scheduling server processes. Neither is the concurrency limited by the degree of multi-threading feasible in the server.
When a process invokes an operation on an object, it carries its thread of execution into the procedure defined in the object. Concurrently running processes thus are able to invoke concurrent operations in an object. This may or may not be allowable, depending upon the nature of the data in an object. The synchronization rules can be effectively built into the object (see section 2.4).

After a process updates the object, the updates may need to be rolled back if the transaction fails, or rather, the updates should be made after the transaction commits. The recovery features that ensure proper atomic updates can also be built into the objects.

The implementation techniques for synchronization and recovery are briefly explained in section 2.4. An outline of a Clouds object is shown in Fig. 2.

The action management system is a part of the Clouds kernel that ensures the atomicity of the nested action scheme. The implementation of the action management system is an integral part of the kernel and is fast, efficient, and runs with very low overhead [Ke85]. Details about the management scheme and action semantics are omitted here.

2.4. Synchronization and Recovery in Clouds

Concurrency control (synchronization) and action atomicity are the responsibility of the objects touched by an action. In Clouds the concurrency control and recovery can be provided by default, by the Clouds system (auto-sync and auto-recovery) or can be tailored for specific objects by the application programmer, or can be omitted altogether for certain objects.

When an object is defined as auto-sync, the object compiler includes default code in the object entry and exit points to adhere to the 2-phase locking protocol. Each operation is classified as read or update operation depending upon the semantics of the operation. Invoking a read operation causes the invoking action to acquire a read lock on the object (if possible, or the process waits until the lock is obtained.) Similarly invoking a write operation causes the acquisition of a write lock, or the upgrade of a pre-existing read lock to a write lock.

Locks are not released when the operation returns or terminates, but are released by the commit phase, which is also handled by the object as described below.
When an action invokes an operation on a recoverable object, the action management system makes a bookkeeping entry to that effect. Each object, by default has three well known entry points, labeled pre-commit, commit and abort. When an action terminates, the action management system invokes these entry points to inform each object that participated in the computation, the result of the action. (The two commit entry points are needed by the 2-phase commit protocol, that is used to terminate successful actions). Activation of the commit or abort entry points in an object releases all the locks.

When a process invokes an object, the object is mapped to virtual memory, and all updates occur in memory. The abort operation simply frees the object, the pre-commit and commit pair flushes all the updates to the permanent storage. [McAl82, PiSp85, Sp84]

2.5. Failure Resilience

Clouds also provides support for single faults, and crashes. A fault generally causes all affected action to abort, providing a guard against inconsistent executions caused by faults. Failure or network partitions can give rise to orphans that hold locks and cause loss of throughput. Clouds supports a time driven orphan elimination scheme that effectively hunts down and aborts all orphaned processes within a short interval after the orphanation [McHe86]. The orphan detection routine makes use of a clever timeout scheme that is rarely affected by system sluggishness. The same scheme with minor modification doubles as a deadlock detector that breaks deadlocks.

The reconfiguration and monitoring system achieves high system resource availability. Since all the users interact with Clouds through an Ethernet, no user is hard wired to a site. A user has a primary site and a backup site associated with the user session. In case of a failure of the primary site, the backup site provides the user with system response. Thus site failures do not cause users to be left without access to the system. Since disks in the system are dual ported, they are accessible from two sites. Only one of the ports is active at any time. If the primary site controlling the disk fails, the other site activates the second port, and effectively all objects located at the faulty site get transferred to the working site. This also enhances the availability of the permanent data in the face of site failures.

The details of the algorithms that control the reconfiguration system are omitted for the sake of brevity.

3. Enhancing Fault Tolerance

The basic fault tolerance mechanism supported by Clouds is the action paradigm implemented by the action management system. The action paradigm ensures consistency of the computing environment in the face of failures. It is a backward recovery scheme. A failed action causes an implicit rollback, and the action may not be able to execute until the fault has been rectified. This degree of fault tolerance can be improved by the usage of better techniques that allow the action to continue using alternate paths of execution.

The key to improved fault tolerance lies in the implementation of a mechanism for the system to monitor itself. The monitoring can be at several levels, discussed later, but the basic components of the monitoring system are probes.

4. System Monitoring and Probes

A distributed system supporting fault tolerance needs a system monitoring subsystem that effectively keeps track of the status of all the hardware components and software components (both active and passive). In this section we present a description of a probe-based monitoring system that can be coupled with the reconfiguration system to enhance the failure resilience of the system.

4.1. Implementing Probes in Clouds

Probes in Clouds are a form of emergency status enquiries, that can be sent from a process to an object or to another process. When a probe is sent to an object, the probe causes the invocation of a probe-procedure defined by default in the object. The probe procedure returns to the caller a status report of the object. This includes the status of the synchronization mechanisms, the actions currently executing in the object and other relevant information (Fig. 3).

Probes can also be sent to processes or actions. A process does not have to explicitly receive a probe. The probe causes a process thread of control to jump to the probe handler, irrespective of the current status of the process. (If the process is currently blocked, it executes the probe handler and returns to the blocked condition.) This is somewhat similar to the way an interrupt causes the CPU of a computer to call an interrupt handler. Thus probes to processes are conceptually similar to interrupts (or signals in the Unix operating system). The probe handler sends a message back to the originator of the probe, reporting the status condition of the process.

The probe handler in the process and the probe procedure in the object are pieces of code that are generated by default by the object compiler or the process compiler, or may be supplied by the programmer of the object or process.

The probe handler/procedures are scheduled and executed at higher priorities than the regular process scheduling priorities. Since the probes cause an immediate, asynchronous reply, and the probes are not suspended by synchronization mechanisms, the time taken...
by the probe to return to the sender is not very dependent on unpredictable conditions, or heavy processing loads. Thus timeouts can be quite effectively used for receiving replies from probes. For example, if the system suspects a faulty network connection from system A to system B, a probe can be sent from system A to the network controller of system B. If the connection is indeed faulty, the probe will not return. System A can effectively determine this condition by timing out, retrying and timing out again. Thus using this scheme we can detect the difference between a dead machine and a slow one, with a high degree of accuracy.

4.2. The Primary/Backup Paradigm

The primary-backup paradigm can be used to enhance the fault tolerance of any crucial system process or action. For instance a critical action that needs to be executed to completion regardless of failures in hardware or communications can be handled as follows.

For each crucial action or process, associate a backup action (or process). The back up action has the same capabilities as the primary, but is dormant most of the time. Periodically the backup process sends a probe to the primary. The primary probe handler sends back an I am OK status report. If the status report is not received in a certain amount of time the backup retries. If the retry fails, this implies the primary is dead, and the backup takes over as the primary and creates a new backup action.

The primary action also periodically checks on the backup. If the backup has failed, then the primary creates a new backup. For obvious reasons the primary and backup should be located at different sites. (Fig. 4.)

The drawback of this scheme is that if network partitioning occurs, we may get two primaries and two backups, as both the primary and the backup decides that the other process is dead. In Clouds however this produces no undesirable effects. Since the objects the two primaries have to access are the same, one (or both) of the actions will fail because of the unaccessability of some object due to the partition. So only one (if any) will run to completion.

This scheme handles single stopping failures. That is if any one process fails the system recovers. After the system recovers, another failure can be handled. But two failures occurring faster than the recovery time cannot be handled. The scheme can be modified to handle a multiple number of failures, but the complexity and overhead increases by a large factor. In Clouds it was deemed very unlikely that two failures will occur in such a short time and not worth the extra expense.

Clouds failure tolerance does not consider corruption or mass failure of storage media. Automatic recovery of storage media can be achieved by mirrored media, stable storage and so on. These methods are well understood, generally expensive and can be integrated into Clouds. We do not consider these techniques at this point.

4.3. System Health Monitoring using Probes

The primary/backup strategy is just one of the applications of the probe system. This scheme only utilizes the ability to send probes to processes. Since probes can be sent to passive objects as well they can be used in more situations. The more powerful application of probes is to build a system monitoring subsystem. System monitoring involves keeping track of the state of each system com-
ponent that has any role to play in the overall function of the system. Passive components such as device interfaces, device control objects and so on, can be monitored by special monitoring processes that invoke diagnostic operations on these entities (or objects encapsulating the device drivers).

The system monitoring subsystem consists of a process (daemon) that runs at each site (monitor). The monitor has a list of components that it needs to keep track of. The list is has a static part and a dynamic part. The static part contains capabilities to various critical system components (network drivers, disk drivers, schedulers, action management system and so on). The dynamic part consists of capabilities to user defined objects and actions that the user expressly records with the monitor, for tasks that require a high degree of fault tolerance.

The monitor at one site has a logical backup, that is a monitor at another site. The various monitors act as primaries for the site it runs on and doubles as a backup for a remote monitor. This allows the distributed system to detect site failures and network partitions. (Fig. 5.)

The monitor periodically probes all the components in its list. The status of these components are stored in a fully replicated database. This database has the same structure and properties as the database used to locate Clouds objects, i.e. it is highly available, but may not be consistent at all sites, or may contain out of date data. The inconsistency of the database does not cause major disruptions in service. The data in the database are used by various system services and the reconfiguration system.

Basically two kinds of failure information is available in the database. The first type is an entry that says a component X at site A is faulty or inaccessible. This is a local fault. The second type denotes that site A is unreachable or dead. This is a fault having global effect. (We are using an Ethernet as a local area network. For point to point communications, more data may be available, e.g. site A is unreachable from site B, but accessible via site C.)

The database updated by the monitoring system ties it to the Clouds reconfiguration system. The reconfiguration system and the interface to the monitor is discussed in the next section.

5. The Reconfiguration System

The reconfiguration system is responsible for maintaining normal system operations in the event of a detectable failure of one or more system components. Clouds recognizes two forms of reconfigurations, namely upwards and downwards. Downward reconfiguration occurs when some component fails, and upward occurs when some new or repaired component is brought on line. We will mainly discuss downward reconfiguration here, as failure is our major concern.

The reconfiguration system in Clouds handles downward reconfiguration by aborting all actions associated with the failed component. It then terminates all orphans.
associated with the failed actions. The users who were affected by the failure are reassigned to functional sites and site search tables are updated to reflect the nonavailability of the malfunctioning site. The failure of a site also makes the objects stored at the site inaccessible. Access to these objects are restored by switching disk ports (if possible).

We divide the task of downward reconfiguration into two parts, namely short term reconfiguration and long term reconfiguration. Short term reconfiguration is used in the case of emergencies or large scale failures. This allows the system to substitute critical components and functionality and keep on running. Long term recovery uses monitoring systems to detect failure of non critical components, intermittent failures or bugs, anticipated failures and such, and tries to isolate faults.

Short term reconfiguration is handled as described above. When a failure is detected, all actions associated with this site and their orphans are aborted. If some of these actions were being run as fault tolerant actions, their backup counterparts would automatically take over. Access to objects that may become inaccessible due to the failure are restored, if possible, through alternate means.

Long term reconfiguration is more refined and adds stability to the system. This incorporates occasional analysis of the distributed (and replicated) directory maintained by the health monitoring system. Any faults or probable faults that may cause disruption are pinpointed and evasive measures taken. The evasive measures include changing the configuration of the system to avoid the usage of suspected hardware and creating alternate paths or alternate services to replace lost components.

Upward reconfiguration is a feature of Clouds that will be useful in most growing environments. This will allow additions of newer hardware, sites and network connections transparently. Upward reconfiguration is handled in two ways. If the new hardware is a device, then its drivers are added as objects, and any application that needs them can access them right away. If a site is added, it simply has to go on line. The search mechanisms will be able to find objects at the new site by its broadcast search mechanism, and the site will automatically enter the search hint files. The whole process is automatic and natural. No special method is needed for upward reconfiguration.
5.1. Interfacing Monitoring and Reconfiguration

The key to a successful reconfiguration system is the acquisition of the failure data. The monitoring system forms a useful source of data for the reconfiguration system.

The reconfiguration system is fully distributed, that is each site runs a reconfiguration daemon. The reconfiguration daemon periodically checks the failure database. (This can be somewhat improved. Details later). When a failure is detected the reconfiguration system works as follows:

If the failure is local, that is the reconfiguration process at site A finds that a component at site A is faulty, it updates the name service databases to this effect. That is this component is marked as unavailable for the system. The reconfiguration system then attempts to get the fault rectified. For example if the component is a device driver, it may delete and reload it from the template and reinitialize it. If the component is a disk, it will attempt to find an alternate path to the disk via a different site, and then transfer all the objects on the disk to the other site. Similar actions can be taken for an unresponsive user workstation.

If the failure is global, e.g. site A is unreachable, it will be detected at another site say B, where the backup was located. At this point the reconfiguration system at site B broadcasts update information to all sites so that no invocation request is sent to site A. It then attempts to transfer the disk connected to A to other sites having ports to A. All users on A are then floated off A to other functional sites.

The conceptual interfaces between the monitor, reconfiguration system, and the object locator database is shown on Fig. 6. The handling of global failures (or site failures) is depicted in Fig. 7.

The periodic scanning of the fault database by the reconfiguration system may not get crash help soon enough. Quick notification of faults to the reconfiguration system by the monitoring system can be achieved by probes. After a critical fault is detected by the monitor, and added to the database, the monitor sends a probe to the reconfigurator, which immediately wakes up and does the needed repair.

The reconfiguration system is of course another fault tolerant system and should be under the supervision of the monitor. However failure of the reconfiguration system cannot be handled by the reconfiguration system and should be handled by the monitor. However failed monitors can be handled by the reconfigurator.

5.2. Long Term Reconfiguration

Most of the techniques described above are applicable for short term reconfigurations. Long term reconfiguration techniques present an interesting field of research that has not been adequately studied. The decisions that the long term reconfiguring system has to take are based on many factors such as importance of the system components, likelihood of failure of questionable components, the hardware system configuration and the availability and usefulness of alternate methods.

For example a lot of system components give rise to intermittent failures. Some of these problems disappear on their own, others work on retries, and then appear again. Especially on device interfaces, these problems can be often solved by reinitializing the device, or the driver.
The monitoring system can be used to keep track of the errors that occur on these components and maintain statistical information on the frequency of occurrences and severity. When errors rise above "normal" levels, the offending component can be rebooted if possible, or marked inaccessible. This can result in early warnings and help in the prevention of system crashes due to flaky components.

6. Debugging Support

As we have stated, the two basic software components of the system are processes and objects. Processes are active entities, executing on behalf of one task (or action). An object is a passive entity, that supports the execution of multiple tasks in its domain.

Several debugging techniques are under study, especially as debugging distributed systems use complex techniques to handle timing and concurrency aspects. Probes can be used to implement a simplistic yet highly effective debugging tool.

We treat processes and objects separately. A special debug probe sent to an object, causes the invocation of an interactive debugging routine, that resides in the object.

This routine allows the programmer to check the insides of an object even when it is in motion. This allows on the fly inspection of system objects without shutting the system down. Depending upon the object, repair on a running engine may be allowable.

A debug probe on a process causes the process to trap to a debug procedure. Similarly, the insides of the process can be checked. However, in this case, the process is not executing user code, and thus only asynchronous snapshots are possible. Some synchrony can then be achieved by setting breakpoints, but the possibilities are not as powerful as the debugging of objects.

7. Conclusions

Clouds is a fault tolerant distributed system. The fault tolerance of Clouds is currently under implementation as an action-based backward recovery system. This type of fault tolerance leaves a lot to be desired. In this paper we have shown an easily implementable subsystem that effectively monitors the status of the distributed system, and actively helps the fault tolerant mechanisms to achieve
forward progress. The paper uses the useful probe mechanism to achieve this and proposes a simple implementation scheme for probes in the Clouds kernel, that can monitor both passive and active system components. Lastly, we present some techniques that use probe based mechanisms for debugging objects and processes.

8. References


[LiSc83] Liskov, B., and R. Scheifler, Guardians and Actions: Linguistic Support for Robust, Distributed Programs, ACM TOPLAS, Vol. 5, No. 3, July 1983


