Distributed Programming with Objects and Threads in the Clouds System*

Partha Dasgupta[†] R. Ananthanarayanan[‡]

Sathis Menon[‡] Ajay Mohindra[‡] Raymond Chen[¶]

[†]Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406
partha@enuxha.eas.asu.edu

[‡]Distributed Systems Laboratory
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
clouds-project@helios.cc.gatech.edu

[¶] Siemens-Nixdorf Information Systems
4 Cambridge Center, Cambridge, MA 02142
chen@sni-usa.com

Abstract

The CLOUDS operating system supports a distributed environment consisting of compute servers, data servers and user workstations. The resulting environment logically simulates an integrated, centralized computing system.

The system paradigm is based on an object/thread model. The basic building blocks for applications are persistent memory (called *objects*) and computation (called *threads*). Unlike most systems, CLOUDS separates the notion of memory from computation. Programming environments based on these abstractions, though unconventional, provide powerful tools for composing applications that exploit concurrency and distribution.

This paper discusses programming techniques that utilize persistence and distribution of memory. The examples show how separation of computation from memory can be used to the programmers advantage. We also present a distributed programming technique called implicit distributed programming. The implementation details of the programming support subsystems are presented. The system performance measurements demonstrate the usability of CLOUDS as a distributed programming platform.

1 Introduction

The CLOUDS operating system supports a distributed environment consisting of compute servers, data servers and user workstations. The resulting environment logically simulates an integrated, centralized computing system.

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

The basic building blocks of a CLOUDS application are persistent memory and computation. Persistent memory is encapsulated in an object specified through a program. Computations are encapsulated by threads which are created at runtime, under user control or under program control. Thus, memory and computation are treated as orthogonal entities. The separation of storage and execution allows concurrent threads in the same address space and allows a thread to execute in multiple address spaces. In addition, persistent memory unifies several levels of the storage hierarchy into a single level store. This system model is called the object/thread paradigm.

In addition, location of an object is orthogonal to the usage of it. That is, the site which provides persistent store for the object need not be the same as the site where the object is being used. This makes applications completely independent of sites. If several computations share the object and execute on different sites, CLOUDS preserves single copy semantics of the object.

Programming environments exploit these features to support an application development platform. Applications built using these environments indicate that the programming paradigm based on object/threads, is more natural to program than comparable applications built using other paradigms. The object/thread model makes CLOUDS an interesting and easy to program system.

1.1 Objectives

In this paper we explore the programming paradigms and techniques that allow implementation of complex distributed applications in a straightforward and simple manner. To this end, we:

- Present the CLOUDS system architecture, the programming paradigm and the vehicles of distribution, namely, Distributed Shared Memory and Remote Object Invocation.
- Provide details on the Distributed C++ (DC++) programming environment (Section 3). This will provide insight into the actual program specification of the features of CLOUDS.
- Show how CLOUDS distributed programs are structured and how CLOUDS' "transparent" distribution facilities are made available to the user (Section 5). In our paradigm, distributed applications can be written in a centralized fashion and yet exploit the parallelism provided by distribution at runtime. This will demonstrate the novelty of the CLOUDS programming paradigm.
- Provide implementation details on the system environment. We will present implementation details of the key operating system features that support the distributed programming environments.
- We reflect on why the separation of processing from memory leads to the building of versatile programming environments. Finally, we contrast it with other systems.

In addition, we present performance measurements that demonstrate the feasibility of our approach (Section 8).

2 An Overview of CLOUDS

The user view of CLOUDS consists of a system architecture and a programming model.

2.1 The Clouds System Architecture

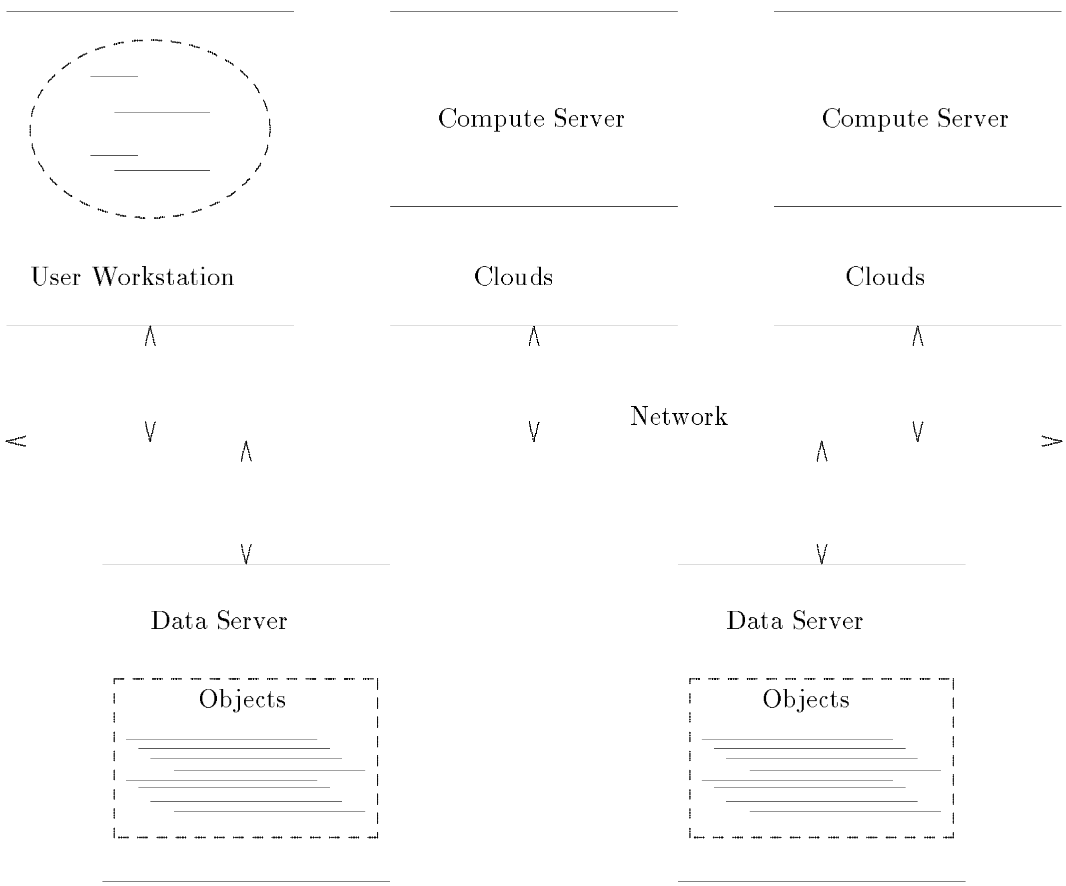

Figure 1: CLOUDS Logical System Architecture

The CLOUDS system integrates a set of homogeneous machines into one seam-less environment that behaves like one, large computer. The system configuration is composed of three logical categories of machines, each supporting a different logical function. These machine categories are *compute servers, data servers* and *user workstations* (see Figure 1).

The core of the system consists of a set of homogeneous machines of the compute server category. Compute servers do not have any secondary storage. These machines provide an execution service for threads. Secondary storage is provided by data servers. Data servers are used to store CLOUDS objects. The third machine category is the user workstation, which are expected to support high-power graphics and other interface capabilities.

The logical machine categories do not have to be mapped to physical machines using a one-to-one scheme. Although a disk-less machine can function only as a compute server, a machine with a disk can simultaneously be a compute and data server. This enhances computing performance, since data access via local disk is faster than data access over a network. However, in our prototype system, we use a one-to-one mapping, in order to keep the system's implementation and configuration simpler.

2.2 Objects and Threads

Objects and threads are the artifacts of programming in CLOUDS. An *object* is a persistent virtual address space consisting of code, data and entry points. Each object is a named instance of a specification (or a class) programmed by the CLOUDS programmer in any language supported by CLOUDS. Once the object is instantiated, it exists "forever", that is until explicitly deleted.

While objects are passive abstractions of memory, a *thread* is an active abstraction of a CPU. A thread starts executing inside an initial object and traverses between objects through *invocations*. On an invocation, the thread leaves the address space of the invoking object and enters the address space of the invoked object. Parameters, if any, are transferred between the invoker and the invoked object on invocation startup and results are returned on termination.

The mapping between threads and objects are defined at runtime by the invocation mechanism. The invocation causes a thread to enter an object. If several invocations occur on an object concurrently, multiple threads will be executing within the object. The invocation mechanisms also allows a thread to execute in multiple address spaces.

Persistent objects provide storage as well as sharing. Files, which are the units of permanent storage in other systems, are not necessary in CLOUDS, since objects are persistent. Similarly the shared memory provided by objects makes message passing unnecessary.

The separation of computation from the address space has far reaching effects. It allows concurrency to be pervasive, that is all code is potentially concurrent. Programs need not explicitly specify concurrency requirements. Also the thread is not tied to its current environment. Using the invocation mechanisms it can reach out to environments that it shared with other threads and applications. This feature is especially appealing when shared and distributed repositories of data is managed and accessed by distributed computations.

Threads execute on compute servers. The permanent storage repository for object content storage are on data servers. Since threads execute within objects, the object being invoked is brought to the compute server running the thread at the time of invocation. The separation of object storage sites and object execution sites allow any thread executing on any compute server $\mathcal{C}$ invoke any object regardless of its storage site. The object will be executed on site $\mathcal{C}$. If two thread $\mathcal{T}_1$ and $\mathcal{T}_2$ executing on compute servers $\mathcal{C}_1$ and $\mathcal{C}_2$ use the same object $\mathcal{O}$, $\mathcal{O}$ will be available at both $\mathcal{C}_1$ and $\mathcal{C}_2$ concurrently. The coherence of the object contents will be maintained by the system.

2.3 Invocations

As mentioned earlier object invocation is the mechanism that makes threads and objects interact. A new thread starts its lifetime when it invokes an object and it terminates when this invocation is over. Thus one thread performs exactly one top-level invocation.

In the course of this invocation, it may perform nested invocations. Each nested invocation is done by a thread upon executing an invoke directive in the program it is executing. The invoke directive is an *external* procedure call (similar to a remote procedure call).

Each nested invocation can be of two basic types:

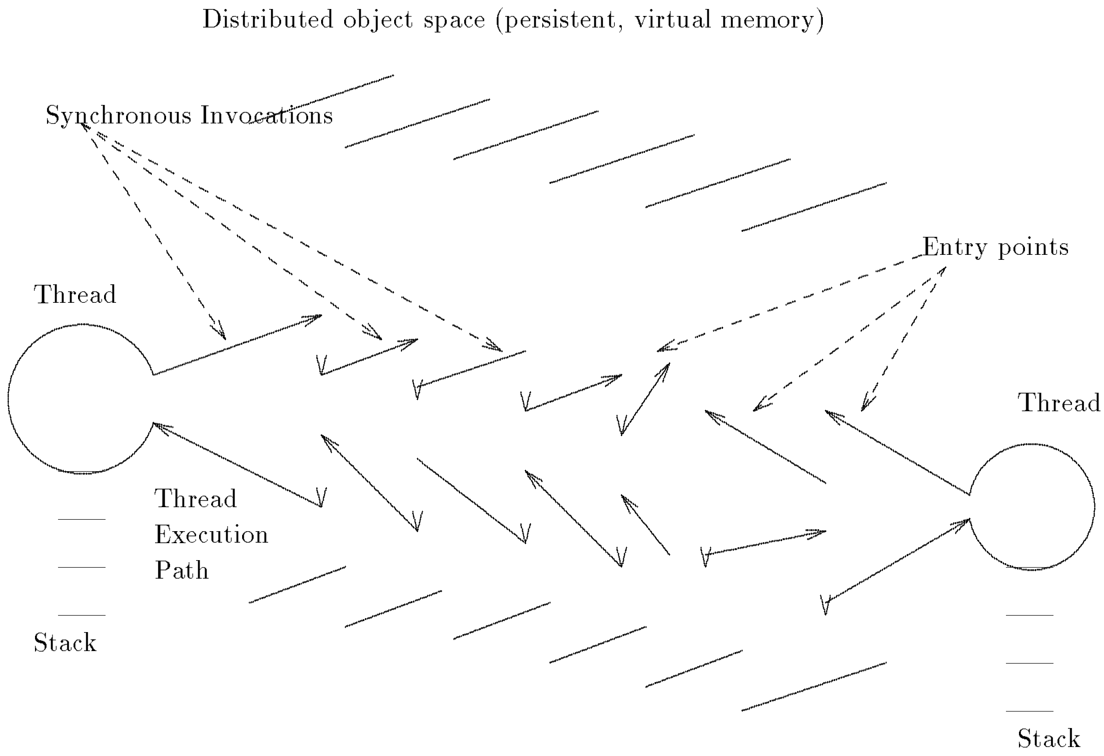

Figure 2: Objects, Threads and Invocations

- A *synchronous* invocation causes thread $\mathcal{T}$ to leave the address space it is currently executing and enter an entry point in another address space (or object). After this invocation terminates, $\mathcal{T}$ returns to the original object as if it has completed a procedure call.
- An *asynchronous* invocation causes $\mathcal{T}$ to create a new thread $\mathcal{T}'$. $\mathcal{T}'$ does a synchronous invocation on the target address space (or object). $\mathcal{T}$ continues to execute in the calling object. $\mathcal{T}$ and $\mathcal{T}'$ continue to execute in parallel, in different objects.

As noted earlier, a thread can invoke any object at any site. This makes CLOUDS appear to be a centralized system where all objects are available at any compute server. To allow distributed programming we allow invocations to have two additional properties.

- A *Local Invocation* causes the target object to be invoked at the same compute server as the thread making the invocation.
- A *Remote Invocation* causes the target object to be invoked at some other compute server. The target compute server may be provided explicitly as an argument to the invocation request, or can be implicitly assigned by the system.

The combination of synchronicity of invocations with the location properties of invocations give rise to four invocation types. These four types of invocations, combined with the concepts of separate persistent memory and threads makes CLOUDS a powerful distributed programming environment. For example the remote asynchronous invocation is a simple but powerful mechanism that is used to start up distributed, parallel computations.

2.4 DSM and ROI

The compute and data servers interact to provide a distributed operating system environment. These interactions occur through the following CLOUDS operating system mechanisms:

- *Distributed Shared Memory* (DSM)
- *Remote Object Invocation* (ROI)

DSM is used to store and share objects in the system. For example, let us assume that a thread is to be run on a particular compute server. The object in which the thread has to execute must be paged from the data server to the compute server. This requires a remote paging facility, which is provided by DSM. DSM supports the notion of shared memory on a non-shared memory, distributed architecture [AMMR90].

In CLOUDS, there is potential for concurrent invocation of the same object by threads at different compute servers. This results in multiple copies of the same object being used at several compute servers. Hence, DSM has to be cognizant of the need to provide the coherence of shared pages.

The coherence specification of an object $\mathcal{O}$ being used at two nodes $\mathcal{A}$ and $\mathcal{B}$, requires that $\mathcal{A}$ and $\mathcal{B}$ see the same contents of $\mathcal{O}$. This is called *one-copy semantics*. The maintenance of one-copy semantics is achieved by coherence protocols that are an integral part of the DSM access strategy [LH86] [RAK89].

Suppose a thread is created on compute server $\mathcal{A}$ to invoke object $\mathcal{O}_1$. The compute server retrieves a header for the object from the appropriate data server[1], sets up the object space, and starts the execution of the thread in that space. As the thread executes in that object space, the code and data of $\mathcal{O}_1$, accessed by the thread, is demand paged from the data server (possibly over the network) to $\mathcal{A}$.

If the thread executing in $\mathcal{O}_1$ generates an invocation to object $\mathcal{O}_2$, the system may choose to execute the invocation on either $\mathcal{A}$ itself or on a different compute server $\mathcal{B}$. In the former case, if the required pages of object $\mathcal{O}_2$ are at other nodes, they have to be brought to node $\mathcal{A}$ using DSM. Once the object has been brought to $\mathcal{A}$, the invocation proceeds. In the latter case, the thread sends an invocation request to $\mathcal{B}$, which invokes the object $\mathcal{O}_2$ and returns the results to the thread at $\mathcal{A}$. More details on object sharing is provided in Section 6.2.

The implication of the CLOUDS DSM mechanism is that every object in the system *logically* resides at every node. This powerful concept separates object storage from its usage, effectively exploiting the physical nature of distributed systems composed of compute servers and data servers.

The second method of interaction between servers in the system is based on the ROI facility. In order to start a user level computation, a compute server must be selected to execute the thread. The selection is controlled explicitly by the programmer, or implicitly by the system based on scheduling policies. The thread is started on the selected compute server by sending a ROI request to the server. The compute server completes an ROI request by obtaining a copy of the object (via DSM) and executing the thread. In addition, ROI

[1] The data is retrieved from the data server that contains the object segments. The system-level name of the object contains the identity of the data server.

is used by threads to distribute computations by initiating further processing on different system compute servers.

CLOUDS ROI is similar to remote procedure call (RPC) mechanisms supported by other distributed systems such as the V system [Che88]. However, it is more general because a CLOUDS ROI can be sent to any machine and the target does not have to store the called object. This capability is a direct result of using DSM as a repository of objects, as discussed above.

To summarize:

- The DSM coherence protocol ensures that objects are globally accessible and data in an object is seen by concurrent threads in a consistent fashion even if they are executing on different compute servers.
- The ROI facility allows for distribution of computation.

The system structure discussed above allows us to support different kinds of structuring of applications, as described in Section 5. In the next section, we give a brief overview of the specific programming environments supported in CLOUDS.

3 The CLOUDS Programming Environments

Currently CLOUDS supports three varied programming environment, namely Distributed C++ (DC++), Distributed Eiffel and CLiDE . DC++ provides a system programming environment based on the C++ language. Distributed Eiffel is an application programming environment based on Eiffel [GL90]. CLiDE is a lisp-based distributed symbolic processing system [PD90]. In this paper we will present programming paradigms for CLOUDS using the DC++ language.

3.1 Basic Programming Environment

The basic environment supports an object-oriented paradigm. The programmer is provided with two kinds of structuring tools: classes (templates) and instances (objects). CLOUDS objects encapsulate particular application behavior and are large grained. A class is the template that is used to generate instances. Object instances may be invoked by user threads. In order to write application programs for CLOUDS, a programmer specifies one or more CLOUDS classes that define the code and data of the application. The programmer then creates the requisite number of instances of these classes. The application is executed by creating a thread to execute the top-level invocation that runs the application.

A user develops CLOUDS programs using the DC++ language and then compiles them on a user workstation. Once compiled, generated objects are automatically loaded onto a data server, making them available to all compute servers. Any compute node (with initiation from a user) can create instances of these classes. Once a class is instantiated, the resulting object becomes part of the persistent object memory and can be invoked until explicitly deleted.

Threads are started in objects either interactively or by explicit thread creation under program control. A user invokes an object by specifying the object, the entry point and the arguments in a CLOUDS *shell* session running on a user workstation. This CLOUDS shell sends an invocation request to a compute server and the invocation commences. User may communicate with the created thread via an X-terminal window on a user workstation.

3.2 The DC++ Environment

DC++ is a programming language and environment that provides support for CLOUDS classes, objects, instantiation, inheritance and naming [Ana91]. DC++ is an extension of the C++ language [Str86]. To give the reader a flavor of programming CLOUDS objects in DC++, we present a simple example.

In this example, we program a CLOUDS class called `rectangle` represented by using the state variables `length` and `width`. The object has two entry points, one for setting the size of the rectangle and the other for computing its area. The class `rectangle` is defined as follows:

```
clouds_class rectangle
    int     length, width;              // persistent data for rect.
    entry   size (int length, width);   // set size of rect.
    entry   int area ();                // return area of rect.
end_class
```

Once the class is compiled, instances may be created. Suppose the `rectangle` class is instantiated, and the instance is called `Rect01`. Now `Rect01.size` can be used to set the size and `Rect01.area` can be called to return the area of this rectangle. The entry point in the object may be called by a command in CLOUDS shell command interpreter. Entry points may also be invoked in a user program, allowing one object to call another.

Objects have user names which are assigned by the programmer when objects are created (compiled or instantiated). In addition, each object has a system-level name (or *sysname*) assigned to it when it is created. SysName are used by CLOUDS to refer to objects and have to be provided to the system when an object is invoked. The resolution of user names to SysNames is performed by the DC++ run-time system, via a name server.

CLOUDS objects are referenced by other programs through variables of a special type defined by the language. The variable type is a class instance reference and is called the *clouds object reference class*. It is represented by the suffix `_ref` appended to the CLOUDS class. For instance, the class that refers to an object of class `foo` is called `foo_ref`.

User level names are *bound* to the sysname of an object before invocations can be performed. This is achieved by the `bind` operation in the reference class. The following code fragment details the steps in gaining access to a CLOUDS object `Rect01` and invoking operations on it:

```
rectangle_ref rect;        // rect is a local program handle that refers
                           // to an object of type rectangle
rect.bind("Rect01")        // call to name server, binds sysname to Rect01
rect.size(5, 10)           // invocation of Rect01
printf("%d", rect.area());     // will print 50
```

The execution of `rect.size` and `rect.area` results in the processing of a local synchronous invocation to the object instance `Rect01`. These operations can be invoked asynchronously by using the syntax `rect!area` or `rect!size`. Remote object invocations (ROI) can be programmed via the virtual node facility (see Section 5.1).

The above example demonstrates programming one CLOUDS object. Since DC++ is an extension of C++, CLOUDS objects can also contain C++ classes and instances. These C++ language entities stored in the address spaces of CLOUDS objects share the properties of CLOUDS objects: they are persistent and can be accessed concurrently via multiple threads that invoke a particular CLOUDS object. Because an object invocation on a CLOUDS object is at least an order of magnitude more expensive than a simple procedure call, a CLOUDS object is appropriate for use as a module that contains several fine-grained entities.

DC++ also provides a variety of other mechanisms to support object programming. These include synchronization, static type checking, built-in data types, memory support services, user I/O support and facilities to define user interfaces. Some of these facilities are outlined in later sections.

User objects and their entry points are typed by the language definition. Static type checking is performed on the object and entry point at compile time. No runtime type checking is done by CLOUDS.

4 Programming A Dictionary

To present the power of combining the building blocks provided by CLOUDS, we present a simplistic implementation of a dictionary. The dictionary is an object supporting the functionalities of insertion, look up and deletion of entries.

The following class defines the dictionary in DC++.

```
  clouds_class dictionary

        recordata[MAX]      // storage for the database

        entry insert(record item);
              // locate index to insert
              lock(data[index])
              data [index] = item;
              unlock (data[index])

        entry record lookup(record item);
              // search the entire array
              // may break up the search into parallel segments.

        entry delete(record item);
              // call lookup then delete using appropriate locks
end_class
```

This implementation of the dictionary stores data as an array in the object. This array is persistent by definition and never needs to be explicitly written out to a file. Also, the usual conversion overhead between internal memory representation and external data format is completely eliminated. Not only does this save a lot of code in the implementation of objects, but also is more natural as the programmer deals with only one type of storage — memory.

An instance of the dictionary may be used by computations by invoking entry points in it. Since these computations can be executing concurrently in the system, the dictionary may be operated upon concurrently by several threads, even if the threads are executing on different compute servers. Note that there is no special program specification necessary on the part of the programmer to achieve concurrency. That is, though the dictionary is a concurrent program, no concurrent programming library or routines are necessary to implement it. However, since all code is potentially concurrent, the programmer needs to use locks to ensure re-entrancy of appropriate portions of the code. On the other hand, the need for locking does not prove to be a problem either. If a programmer does not want concurrency, the compiler can be instructed to protect all entry points with a lock, like in a monitor.

The `lookup` entry point does not modify the data contained in the dictionary. Consequently, when threads at several sites invoke the `lookup` entry point, the dictionary gets *replicated*, automatically. There is no replication code specified in the program itself. The DSM mechanism provides replicated copies to all users of the dictionary, if the `lookup` entry point is the only one invoked. This is done at runtime with *no hints* from the programmer (that is the `lookup` function was not defined to be a read-only function). Hence, operations on an object which do not modify the data in it can be executed completely in *parallel*, even though the computation is performed on different sites. However, note that this form of replication enhances the performance of the system, but not the fault-tolerance.

When a thread at any site invokes an entry point that modifies the dictionary (for example, the `delete` entry point), the page containing the data being modified is automatically *yanked* (invalidated) from all replicated copies of the data. Later, if any thread reads that page, the new (updated) copy is automatically provided. Note that this automatic replication is a due to DSM and is not programmed by the user.

The example shows some of the ease of programming in CLOUDS. Concurrency, read-replication, transparent distribution and persistence are all utilized in this simple program. The programmer does not have to explicitly program these features into the objects, but they are inherited from CLOUDS.

5 Distributed Programming in the CLOUDS System

While CLOUDS programming environments provide a means to specify application programs, the structure of these application can vary widely. As noted earlier, in CLOUDS, objects can be written to run on a centralized computing environment without regard to concurrency, replication, etc. In contrast, consider a message based distributed system. One of the popular means of structuring a distributed program is as set of processes consisting of a master and a number of slaves. The master allocates work to the slaves which perform the necessary computation and send back the result. All communication is done through messages. In CLOUDS, the same effect can be achieved in a different and manner that is easy to understand and program. This section discusses how programmers can organize such an application.

An application program, consisting of a set of classes and objects, can be structured in three different ways:

- Treat the CLOUDS system as one integrated, centralized system. Each application is programmed as one or more CLOUDS classes, each with one or more instances. Existing classes can be reused. Each instance is an object that can contain a complete C++ object oriented environment. This is *centralized* programming.

- The programmer can also decide to use the CLOUDS system as a distributed system in which each object is a pseudo-node. Computations are executed in as many nodes as there are objects. This is *explicit* distributed programming.

- The third alternative is to structure the application as one (or more) object(s) as in the centralized scheme, but execute the computation in a distributed manner by starting the computation at several nodes, regardless of where the objects are located. This is called *implicit* distributed programming.

In addition to the above, the programmer can exploit the persistent nature of the objects as well as utilize the concurrency within each object. Centralized programming is the same as traditional sequential object oriented programming and will not be dealt with here. We also do not present persistent and concurrent programming paradigms in this paper. In the rest of this section, we discuss how explicit and implicit distributed programming can be achieved.

5.1 Virtual Nodes

The CLOUDS *virtual node* facility is designed to let users target computations to particular virtual nodes. The system is treated as a set of virtual nodes, each having a node number within a sequential range of integers. The programmer, while coding CLOUDS objects, is unaware of the actual physical configuration of the system. Programs request the desired number of nodes from the run-time system associated with the virtual nodes facility. If the system can satisfy this request, it returns the actual number of virtual nodes available to the programmer. The program then partitions its computation based on the number of nodes granted. The number of virtual nodes in the system need not correspond to the number of physical nodes.

To run an invocation on a particular node, the user provides the invocation request with a virtual node number. For example, to synchronously invoke the operation `op` on object `O` on a virtual node identified by `node_num` (exact syntax not shown for brevity):

```
O.op (params) at node_num;
```

Similarly, an asynchronous invocation to `op` on object `O`:

```
O!op (params) at node_num;
```

5.2 Explicit Distributed Programming

The CLOUDS system can be used as a traditional distributed programming system. The unit of distribution is the object. The programmer decides on the number of objects by analyzing the characteristics of the application. For example, consider distributed sorting. One possible algorithm creates `n` sorter objects, one on each virtual node. Then, the data is partitioned into `n` parts and sent to each of the sorter objects which sort the data and return the results to the main computation. The main computation then merges the `n` sorted pieces. This is what we call explicit distributed programming.

Programming an arbitrary algorithm in this fashion is very similar to programming clients and servers in a distributed message system. It involves explicit programming of the distribution and protocol definition to be used for client-server communications and intricate algorithm development. The degree of distribution is also *statically* defined by the program, since the number of objects is fixed.

5.3 Implicit Distributed Programming

In CLOUDS, using DSM and the different types of invocations[2] it is possible to program distributed applications without using the client server model. This allows distribution to be expressed implicitly, and provides the ability to make decisions on the degree of distribution at runtime.

In implicit programming, the application is structured as one centralized application, typically using *one* CLOUDS object. The unit of distribution is the thread. Implicit distributed programming structures the program as a concurrent program and not a distributed one. Each thread in the concurrent execution runs on a different virtual node, but uses the same object(s). Since DSM provides one-copy coherent memory across machines, the computation will actually work like a concurrent program. However, if the concurrent threads do not heavily share the same pages of memory, the performance will be similar to an explicitly distributed program.

We present a distributed sorting algorithm based on the above idea. The following program implements *one* object called `sorter`, that contains an integer array, which is to be sorted. The object has an operation to sort the array (the entry point `sort`) which is invoked when the data needs to be sorted. The code implementing the `sort` operation partitions the computation using virtual nodes.

The operation `subsort` is another entry point in `sorter` that sorts part of the array based on parameters which specify starting and ending points in the array. `subsort` is executed at different virtual nodes depending on the number of virtual nodes available at runtime. When all the `subsorts` terminate, the array is merged.

```
clouds_class sorter
      entry sort(); // sort the entire array
private :
      int array[MAX];
      entry subsort(int i, int j);
end_class

sorter::subsort(int i, int j) {
      // sort from array[i] to array[j], in place.
}

sorter::sort() {
      // Get N nodes from the virtual node facility
      int no_nodes = N;     // number of nodes.
      int seg_size = MAX/N; // can be changed dynamically.
      for (int node = 0; node < no_nodes; node++) {
```

[2] Synchronous and Asynchronous combined with Local and Remote.

```
                sorter!subsort(node * seg_size,
                         (((node + 1) * seg_size) - 1)) at node;
            }
            //Wait for invocations to terminate; merge sorted segments
    }
```

The sub-sorts are concurrently executed using asynchronous invocations. Thus, the sort is executed by multiple threads which execute at a different (logical) compute servers, and perform computation on different parts of the data in parallel. Note that the data itself is encapsulated in a *single* object. The data actually required by each thread migrates to that node automatically, via DSM, as discussed in Section 6.2.

Therefore, programming of this sorter object is achieved without explicit distribution of data, or any knowledge of the actual distribution of the algorithm. Decisions concerning the degree of distribution of the algorithm are made at runtime.

6 The Implementation of the System Environment

CLOUDS is implemented as a native operating system on Sun-3 computers. The compute servers run CLOUDS. The data servers and the user workstations are implemented by server processes on UNIX workstations.

CLOUDS is hosted by a minimal kernel called *Ra*. Ra provides the basic memory management and scheduling mechanisms. CLOUDS is built on top of Ra by using pluggable system service modules called *system objects*. In this section we will discuss the system objects that provide support for distributed programming: the invocation system, the synchronization system and the DSM system. In addition, we discuss user-level utilities that provide compilation support for user objects. A more comprehensive description of the implementation of CLOUDS is available in [DCM+90] [DJAR91].

6.1 The Invocation System

Objects in CLOUDS are implemented as shared virtual address spaces. Each object has an object header that defines the layout of the object address space. Threads are implemented using local processes. If a thread executes on only one node, then it will be associated with only one process. However, if the thread performs remote object invocations then the thread will have multiple processes executing on behalf of the thread; one on each machine touched by the distributed thread.

A thread executing in one object invokes another object through a system call. The Invocation System then determines from the system call parameters whether the invocation is to be asynchronous or synchronous, and whether it is a local or remote invocation.

In the case of a synchronous local invocation, the state of the current object invocation is saved. Next, using information in the header of the invoked object the new object is installed into the address space of the executing thread. When the thread resumes execution, it will be executing in the address space of the new object. Asynchronous local invocations are implemented by creating a new thread to perform the object invocation.

Synchronous remote object invocations are implemented using slave processes on the remote site and is similar to conventional RPC implementation [BN83]. In the case of an asynchronous remote invocation, the invoking thread does not block.

6.2 Paging and Sharing of Object Code and Data

The DSM system is responsible for making all objects available to all compute servers. It is the software layer between the demand paging system of the RA kernel and the storage daemons running on data servers. The DSM system has several subsystems, namely: *DSM Server* and *DSM Client*. Each compute server includes a DSM client and a DSM server. The data servers each run a DSM server as a Unix process. The communication transport protocol used to communicate between the corresponding system components in different machines is called RaTP (Ra Transport Protocol) [Wil89].

Suppose a compute server $\mathcal{A}$ running a computation faults on page p of data. This fault activates the DSM Client by generating a call to a method in the system object. The DSM Client locates the DSM server containing page p. The server, called the *owner* for any particular page is fixed, systemwide.

Let site $\mathcal{D}$ be the owner of page p. The DSM Client on site $\mathcal{A}$ sends a request to the DSM server on $\mathcal{D}$. If p is currently not being used by any other compute server, $\mathcal{D}$ sends p to $\mathcal{A}$ and the computation progresses. Site $\mathcal{A}$ now becomes the *keeper* of p.

At this point, suppose another computation on another site $\mathcal{B}$ page faults on the same page p. $\mathcal{B}$ sends a request to the owner, $\mathcal{D}$. $\mathcal{D}$ forwards the request to the DSM server on $\mathcal{A}$, since $\mathcal{A}$ is the keeper of p. In response to the forwarded request, the DSM server at $\mathcal{A}$ unmaps p from the address space of the thread using the page and sends it directly to $\mathcal{B}$. This is called *yanking* the page. If both $\mathcal{A}$ and $\mathcal{B}$ use a page concurrently, this page will *shuttle* between $\mathcal{A}$ and $\mathcal{B}$ guaranteeing one-copy semantics [LH86] [RAK89].

In the above scheme, each page has one owner (the data server) and at most one keeper (the compute server using it). For read-only pages the constraints are relaxed, and a page can have multiple keepers. Read-write pages can be acquired in read-only mode (via read-mode page faults) allowing better performance when pages are read-shared by several compute servers.

6.3 Support for Synchronization

The data space of an object is shared by all computations that execute in the object. Since computations can run in an object concurrently, there is need for mechanisms that provide mutual exclusion and thread synchronization. The data in an object is accessible only by threads executing within the object. Hence, programming of thread synchronization is local to each object. However, the same object may be used by concurrent threads running on different compute servers. Thus, the synchronization must work across machines. This section discusses the implementation of semaphores and locks that provide intra-object, distributed synchronization.

Synchronization support can be provided at the language level using constructs such as semaphores and monitors. The implementation of such constructs, however, needs operating system level support. CLOUDS provides support for synchronization in the form of semaphores and read write locks [Ana]. Each semaphore or lock is identified by the CLOUDS operating system by a name that is composed of two parts: a sysname and an instance identifier. This scheme eases management of these lock names by imposing a logical hierarchy, based on their intended use. The sysname can be the same as the sysname of the object where the semaphore/lock is defined, and each semaphore/lock within the object has an instance identifier. All state information associated with semaphores and read-write locks is maintained by the operating system.

Semaphores support *create*, *P* and *V* operations. Read-write locks support locking in *read* mode or *write* mode, and unlocking. In addition, a *get* operation is provided with both semaphores and read-write locks. The *get* operation is a directive to cache the state information corresponding to a particular synchronization primitive at the node executing the operation. This operation can be used to improve performance by making use of locality of access to the semaphore or the read-write lock.

6.4 From Programs to Objects

In this section, we briefly describe how objects are created from a program specification. In particular, we discuss the implementation of DC++.

DC++ programs are developed on user workstations and are stored as Unix text files. A DC++ program module consists of a class definition file and an implementation file. These programs are converted to C++, using a preprocessor. The converted programs define a CLOUDS class. In addition, the preprocessor generates interface stubs to access this class. These include the CLOUDS object reference class (See Section 3.2) and the information needed to support inheritance of CLOUDS classes. All this information completely defines a CLOUDS class and is stored as part of the environment of the programmer. This environment serves as a library when that CLOUDS class is used or inherited by other CLOUDS classes. C++ programs are compiled with a standard compiler along with the DC++ library which defines, among other things, the CLOUDS system call stubs.

After the compilation of the program(s) to Unix `.o` files, the programs are linked with the DC++ library using the UNIX link editor (`ld`). This creates a UNIX executable with the `a.out` format. The `a.out` file is then post-processed into segments that adhere to the CLOUDS object format[3]. The program is now stored as two files containing the data segment and the code segment.

The segment files are then loaded on the CLOUDS data server. This is accomplished by adding the segments and the object descriptor (another segment) to the list of segments managed by the data server. At this point, the segments are accessible on the CLOUDS system. Objects represented by these segments can then be invoked or instantiated.

7 More Programming Support

In addition to the programming support mentioned in earlier sections, the CLOUDS system supports various types of persistent memory and provides consistency support for persistent objects. These allow CLOUDS programs to use advanced memory structures and define consistency requirements of applications.

7.1 Memory Semantics

Persistent memory needs a structured way of specifying attributes such as longevity and accessibility for the language-level objects contained in CLOUDS objects. To this end we provide several types of memory in objects. The sharable, persistent memory is called per-object memory. We also provide per-invocation memory that is not-shared, but is global

[3] An object may contain multiple data segments. The layout and number of segments are under the control of the programmer.

to the routines in the object and lasts for the length of each invocation. Similarly per-thread memory is global to the routines in the object but specific to a particular thread and lasts until the thread terminates. This variety of memory structures provides a powerful programming support in the CLOUDS system [DC90].

7.2 Consistency Support

The CLOUDS *consistency-preservation* mechanisms present a uniform object-thread abstraction that allows programmers to specify a wide range of atomicity semantics. This scheme performs automatic locking and recovery of persistent data. Locking and recovery are performed at the segment-level and not at the object level. Since segments are user defined, this allows the user to control the granularity of locking. Custom recovery and synchronization are still possible but will not be necessary in many cases.

Threads are categorized into two kinds, namely *s-threads* (or *standard* threads) and *cp-threads* (or *consistency-preserving* threads). The s-threads are not provided with any system-level locking or recovery. The system supports well defined automatic locking and recovery features for cp-threads. When a cp-thread executes, all segments it reads are read-locked and the segments it updates are write-locked. On completion, the segments are committed and locks released. Further, cp-threads are classified to support *global* consistency across objects and *local* consistency within an object. Since s-threads do not automatically acquire locks, nor are they blocked by any system acquired locks, they can freely interleave with other s-threads and cp-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 [CD89].

8 Performance

This section presents performance measurements for the invocation subsystem and other related subsystems in CLOUDS, which supports the programming environments outlined in the paper. In our environment, compute servers run on *diskless* Sun-3/60 machines; data servers and user workstations are Sun SPARCstation 1 machines running UNIX.

Kernel Operation	Time
Page Fault Service (Local) without Zero Fill	629 μs
Page Fault Service (Local) with Zero Fill	1.5 ms
Page Fault service from data server (Remote)	16.3 ms

Table 1: Basic Timings

Object invocation involves the paging in of the object header from the data server and the installation of an address space which contains the objects text and data, from the information contained in the object header. When the threads starts executing in the newly installed address space, the text and data are fetched on demand by the page-fault handler, in co-operation with DSM. The basing timings for page-fault handling, when the page is resident on the same node costs 1.5 ms for a zero-filled 8K page and costs 629 μs for a non zero-filled page. Such faults do not require network messages.

The time taken to service a page fault, which requires the page to be fetched from a remote data server, costs 16.3 ms. The page fetch over the network uses the RaTP reliable transport protocol.

Invocation Operations	**Time**
Synchronous Local Object Invocation	
- 1^{st} time	93 ms
- 1^{st} time, 1 data page	119 ms
- 2^{nd} time	8.9 ms
Asynchronous Local Object Invocation	
- 1^{st} time, return from call	66 ms
- 2^{nd} time	17.8 ms

Table 2: Invocation Performance

Table 2 summarizes the costs for local object invocation. Invoking an object for the first time involves at least two page-fault operations for bringing the object header (an 8K page) and one page of code. Such an invocation takes 93 ms., while an invocation that also accesses a data page takes 119 ms.

The next time the same object is invoked, its pages are cached in memory and invocation time (8.9 ms.) drops sharply. This is because no page fault occurs and no network network access is needed. Overhead in this case involves switching the address spaces of processes. In general, object invocation costs should be amortized over the lifetime of the object at a particular compute site.

A local asynchronous invocation is measured from the time the invoking thread issues the invocation request to the point the request returns. This involves setting up the object header (paging in one page, on the first invocation) and creating a new thread. The total time of 66 ms does not involve bringing in code or data pages or waiting for the newly created thread to run. The new thread waits for its time slice before it executes and may wait a long time before it actually executes. Costs for subsequent invocations is less since the object header mapping does not involve network access. However, its cost is larger than a synchronous invocation due to the thread creation overhead.

Remote invocations are almost identical to the local invocations, except that an invocation request is sent to another compute server.

The performance measurements for the CLOUDS distributed operating system show that it is quite competitive with any system that works over a network without local disks. While initial operations are slower, subsequent operations are considerably faster. Thus, the speedup of subsequent operations due to caching provides fast overall execution characteristics when network costs are properly amortized.

9 Related Work

Distributed programming has been around ever since networking was made possible. Some of the first major distributed applications such as *uucp* and *UseNet* used handshaking over communication lines without operating systems support. Bal et. al. [BST89] presents a comprehensive survey of programming languages and systems developed for distributed system, classifying them by functionality and intent. A complete discussion of all the environments is beyond the scope of this paper, and we shall compare CLOUDS to some of the closely related systems.

Orca is a programming language and runtime system to program distributed applications [BKT90]. It extends the abstract data type model to distributed systems through *shared data objects*. The runtime system of Orca provides support for sharing and location of objects. The programming support is heavily dependant on the Orca runtime mechanisms and not the underlying operating system. In contrast, CLOUDS provides most of the support necessary for DC++. This allows multiple language support possible without reimplementing the runtime support for each language. As mentioned before, Distributed Eiffel and CLiDE are two other environments that run on top of CLOUDS. Distributed Eiffel provides a more structured programming environment and is intended for casual programmers while DC++ is intended for systems programmers. CLiDE is an environment implemented using DC++ and caters to symbolic programming needs, such as those of AI applications. While this allows for the user to choose an appropriate vehicle of expression depending on the application, effort is not duplicated in implementation of multiple runtime systems that perform similar tasks. Further, object sharing in Orca is restricted to processes that are related. Such a restriction does not arise in CLOUDS as a direct result of persistence of objects and orthogonality of computation and objects.

While Orca hides the location of objects from the programmer, Emerald [JLHB88] provides mechanisms to move objects when necessary. This feature is similar to CLOUDS, where objects move to a node on invocation and remain there if no other nodes invoke the object. However, automatic replication due to immutable invocations are handled differently. Replication is controlled in Emerald by the programmer by claiming the object to be immutable: the system does not check the validity of the claim. While this is potentially dangerous due to possible programming errors, it is also a *static* property. In CLOUDS, on the other hand, replication is controlled *dynamically* based on actual usage as illustrated in the dictionary example.

Some operating systems implement their own versions of objects at the kernel level. These include Argus [Lis84], Cronus [STB86] and Eden [ABLN85]. The objects in these systems are modules that run as Unix processes and respond to invocations or messages. The objects can be checkpointed to files on demand. Lightweight threads are used to provide intra-object concurrency and the threads are handled by built in libraries. Unlike CLOUDS these systems do not provide the orthogonality of computations and memory making programming not as elegant.

The Commandos operating system [MG89] provides support for all types of objects (fine and large grained, persistent and volatile) in an uniform fashion. The operating system provides management of object types, location, and sharing. Among other things, since object typing is directly supported by the operating system, Commandos is closely tied to the programming environment and thus is a special purpose, single paradigm system.

In Mach [ea86], multiple threads share a task, which is the unit of sharing and protection. All resources of a thread are accessible to other threads associated the task. At a programmer's level, concurrency is explicitly programmed by creating threads using a library package. Sharing is also possible through memory objects, which have to be explicitly mapped in into a task's address space. In CLOUDS, this is automatically done on invoking the desired object.

The memory in most of the above systems are private to the process or application using the objects. The objects exist in the global address space of the process executing the application and are copied in or out of the space, as and when necessary. When the process terminates, the memory is lost and the persistent objects have to be saved on secondary storage. Thus the objects, when shared exist in the memory space of multiple processes. Our approach is the opposite. The object does not appear in multiple address spaces. The threads visit the objects address space. We feel that this approach is cleaner, easier to program, comprehend and also easier to implement.

10 Conclusions

The support for programming distributed objects in a variety of programming languages and environments is one of the strong points of the CLOUDS distributed operating system. The system provides persistent objects that can be used for programming applications. Since the objects are persistent, there is no need for explicitly saving state. In fact, the operating system does not provide for file systems or disk I/O routines available from the user environments. In addition, CLOUDS distribution mechanisms allow the programmer to implement applications using implicit distribution techniques. Coupled with the orthogonality of compute and data servers, the system design is elegant, easy to use and intuitive. This enhances its usability and represents the novel aspect of the CLOUDS system environment.

The performance of the system is more than adequate. The compute performance is dependent on the machines used to run the applications; the only bottleneck being the paging of the objects from the data servers. This can be improved with high speed networks or placing the data servers on the same machines as the compute servers. However, keeping the data servers physically separate has some distinct advantages: orthogonality, uniform access costs and symmetry. We could improve local performance by integrating the compute and data server functions together on one machine. However, this approach would not improve global system performance, since objects located at the combined compute/data server node would still have the same remote access characteristics as they did when the functions were separated. Instead, a solution involving a high-speed network appears more favorable. Thus, in most cases (except if the host is a high-power multiprocessor) the data servers should be kept separate and linked via a high speed network.

11 Acknowledgements

We thank Mark Pearson for working on the earlier version of this paper. We also thank Chris Wilkenloh, Gautam Shah, Vibby Gottemukkala and M. Chelliah for implementing some features of the system.

References

[ABLN85] G. Almes, A. Black, E. Laswoska, and J. Noe. The Eden System: A Technical Review. *IEEE Trans. on Software Engg.*, SE-11, January 1985.

[AMMR90] R. Ananthanarayanan, Sathis Menon, Ajay Mohindra, and Umakishore Ramachandran. Integrating Distributed Shared Memory with Virtual Memory Management. Technical Report GIT-CC-90/40, Georgia Institute of Technology, 1990.

[Ana] R. Ananthanarayanan. An Implementation Architecture for Synchronization in a Distributed System. Technical Report (in progress).

[Ana91] R. Ananthanarayanan. CC++ Reference Manual. Technical Report GIT-CC-91/07, Georgia Institute of Technology, College of Computing, Distributed Systems Laboratory, 1991.

[BKT90] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Experience with distributed programming in Orca. In *In Intl. Conf. on Computer Languages*, 1990.

[BN83] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, October 1983.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Comuting Surveys*, September 1989.

[CD89] Raymond C. Chen and Partha Dasgupta. Linking Consistency with Object/Thread Semantics: An Approach to Robust Computation. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, June 1989.

[Che88] D.R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–33, March 1988.

[DC90] Partha Dasgupta and Raymond C. Chen. Memory Semantics in Large Grained Persistent Objects. In *Proceedings of the 4th International Workshop on Persistent Object Systems (POS)*. Morgan-Kaufmann, September 1990.

[DCM+90] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu-Aubán, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The Design and Implementation of the *Clouds* Distributed Operating System. *Usenix Computing Systems*, 3(1), 1990.

[DJAR91] P. Dasgupta, Richard J. LeBlanc Jr., Mustaque Ahamad, and Umakishore Ramachandran. The CLOUDS Distributed Operating System. *IEEE Computer*, April 1991. *To appear.*

[ea86] M. Accetta et. al. Mach : A New Kernel Foundation for Unix Development. In *Proc. Summer Usenix*, July, 1986.

[GL90] L. Gunaseelan and R. J. LeBlanc. Distributed Eiffel: A language for programming multi-granular, distributed objects. Georgia Tech Distributed Systems Laboratory, *Submitted for publication*, October 1990.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb 1988.

[LH86] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proc. 5th ACM Symp. Principles of Distributed Computing*, pages 229–239. ACM, August 1986.

[Lis84] B. Liskov. Overview of the Argus Language and System. Technical Report Programming Methodology Group Memo 40, M.I.T., Labaratory for computer Science, February 1984.

[MG89] J. A. Marques and P. Guedes. Extending the Operating System Support for an Object Oriented Environment. In *In Proc. OOPSLA-89 Conference*, October 1989.

[PD90] M. Pearson and P. Dasgupta. CLIDE: A Distributed, Symbolic Programming System based on Large-Grained Persistent Objects. Technical Report GIT-CC-90/62, Georgia Institute of Technology, College of Computing, Atlanta, GA., November 1990.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronizat ion and Data Transfer. In *Eighteenth Annual International Conference on Parallel Processing*, August 1989.

[STB86] R. E. Schantz, R. H. Thomas, and G. Bono. The architecture of the Cronus distributed operating system. In *Proc. of the 6th Int'l. Conf. on Distr. Computing Sys.*, May 1986.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 1986.

[Wil89] Christopher J. Wilkenloh. Design of a Reliable Message Transaction Protocol. Master's thesis, Georgia Institute of Technology, College of Computing, 1989.