[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.

[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.

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.

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

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

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.

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.

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.

```
            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.

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 `subsort`s terminate, the array is merged.

clouds_class `sorter`
    entry `sort`