CLiDE: A Distributed, Symbolic Programming System based on Large-Grained Persistent Objects *

Mark P. Pearson
Distributed Systems Laboratory
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
mpearson@cc.gatech.edu

Partha Dasgupta
Department of Computer Science
and Engineering
Arizona State University
Tempe, AZ 85287-5406
partha@enuxva.eas.asu.edu

Abstract

Clouds LISP Distributed Environments (CLiDE) is a distributed, persistent object-based symbolic programming system being implemented on the Clouds distributed operating system. LISP environment instances are stored as large-grained persistent objects, enabling users on many machines to share the contents of these environments through inter-environment evaluations. CLiDE provides a comprehensive research environment for distributed symbolic system language, invocation and consistency semantics and an implementation vehicle for the construction of the symbolic processing portions of complex *megaprogrammed* systems.

1 Introduction

In the past, distributed system research has primarily focused on the theory and implementation of basic mechanisms and policies involved in distributed system construction, but has largely failed to address the actual programming paradigms, administration, features and use of such systems. Shortfalls of current "networked" computing systems, a greater understanding of fine-grained object-oriented program construction and the existence of experimental distributed systems has motivated research in distributed systems programming techniques and programming in the large.

This paper presents the design of CLiDE, a distributed symbolic programming system. CLiDE is built on top of the Clouds distributed operating system which is currently in use at Georgia Tech. CLiDE allows users to construct large distributed programs via the specification and instantiation of distinct, but sharable, symbolic processing environments. These environments, which encapsulate particular behaviors or services, are stored as large-grained, Clouds objects which persist until explicitly deleted. The motivations for constructing distributed symbolic programming systems and the current state of such systems are presented in Section 2. We describe design goals of the CLiDE system in Section 2.1. In Section 3, an overview of the Clouds distributed operating system is presented. The design of CLiDE is presented in Section 4 and is followed by a discussion of the usage of CLiDE in Section 5. The *megaprogramming* paradigm as it relates to CLiDE and the implementation of distributed applications is presented in Section 6. Finally, the CLiDE system's current implementation status and future research directions are presented in Section 7.

2 Motivations and Related Work

Distributed problem solving through the use of multiple cooperating subsystems, experts, or agents has been a focus of recent research in the computing sciences. Examples include autonomous mobile robot navigation [2], distributed decision support systems [6] [12], multiple heuristic systems and Distributed Smalltalk applications involving heterogeneous, non-Smalltalk processors [3] [4].

Recently, research in Distributed Artificial Intelligence (DAI) has focused on the construction of software architectures which support the interaction of multiple intelligent systems. Multiprocessor oriented DAI architecture work has primarily focused on system performance and parallel execution. The Cage system allows single users to specify parallelism for multiple computing agents on a shared memory multiprocessor [1], while the Poligon system supports concurrent, blackboard-based AI applications on distributed memory multiprocessor machines with high-bandwidth interprocessor communication [21]. Mace (Multi-Agent Computing Environment) allows each instantiated agent to run on a specific processor of an Intel Sym-1 large-memory hypercube [12]. In all of these systems, the distribution of cooperating agents is simulated on a single multiprocessing system.

Simulact *simulates* a multiprocessor system with a one to one agent/processor mapping on a network of LISP machines [19]. In Simulact, each agent runs synchronously with all the other agents in the system via a central coordinator. Despite the distributed nature of their applications, Cage, Poligon, Mace, and Simulact support cooperating agents in a *single-user, global environment.*

Distributed Smalltalk provides a framework for cooperation among geographically separate Smalltalk users by allowing direct access to remote, small-grained Smalltalk

*This work funded by NSF grant CCR-86-19886.

objects and, to some extent, environment sharing. Each user environment exists in a logically distinct address space on a specific machine. The complexity of the Smalltalk class hierarchy limits object mobility between machines.

Avalon/Common Lisp extends Common Lisp [23] to support fine-grained, recoverable atomic objects and remote evaluation [22]. This implementation allows remote evaluation of LISP expressions among a group of distributed evaluators and supports limited transparent object transmission between evaluators. Remote evaluations involving complex data types require the user to define sending and receiving object transmission functions. Each evaluator process is single-threaded and has a private *recoverable store* managed by a separate process for recoverable, atomic objects [9].

Persistent CLOS (PCLOS) is an extension of CLOS [5] that supports database independent persistence of fine-grained CLOS objects beyond the LISP session that created them [20]. Through PCLOS language constructs, users access a virtual database that contains CLOS object information. This virtual database is physically implemented by several database servers which communicate over an Ethernet.

Distributed Smalltalk, Avalon/Common Lisp, and PCLOS are distributed implementations based on communicating processes that may not migrate. As a result, the host operating system for these implementations is unable to perform load balancing operations on processing activity pertaining to their execution. PCLOS provides distribution of CLOS objects and methods but does not support remote evaluation. Problems with environment security in Distributed Smalltalk allow "users to shoot others" [3]. It is not apparent that Avalon/Common Lisp addresses these security issues either; instead they focus on recoverable atomic objects.

2.1 CLIDE Design Goals

The distributed symbolic processing system we envision is one that:

Provides mechanisms for cooperation among multiple users and seamless sharing of distinct environments. Seamless sharing of environments allows straightforward construction of distributed decision support systems.

Supports environment persistence. Activities pertaining to the saving, loading, and initialization of a particular user's symbolic processing environment are time-consuming. If environments can persist, then the time devoted to these activities can be eliminated.

Provides mechanisms for parallel execution within a distinct environment. Intra-environment parallelism allows previous work on efficient parallel LISP evaluation to be extended to multiple site distributed systems.

Provides mechanisms for maintenance of environmental consistency and security. These qualities are necessary for any system which must maintain consistency of shared environments for a community of users.

Supports the architecture of future megaprogrammed information systems.
Some of the above features can be achieved by using the large-grained persistent objects and threads provided by the CLOUDS operating system. Portions of the design of CLIDE were inspired by the availability of these features. Since the design of CLIDE depends upon the semantics of CLOUDS objects and threads, we first present a brief overview of the CLOUDS distributed operating system.

3 CLOUDS Overview

CLOUDS is a general purpose, experimental, distributed operating system which supports two simple primitives: threads and persistent object memory. With these primitives, CLOUDS integrates a number of loosely-coupled machines to produce a system that behaves like a centralized, time-sharing system [11].

3.1 Object and Threads

A CLOUDS object is a passive, persistent virtual address space that is not associated with any process or thread. In terms of conventional operating systems, CLOUDS objects are persistent like files; differ from active, process-associated virtual address spaces; and are accessed like addressable memory.

Like fine-grained objects supported by object-oriented languages such as Smalltalk [13] and C++ [24], CLOUDS *large-grained objects* contain *data* and *methods* that operate on the data. The large size of CLOUDS objects and the overhead associated with object invocation favor them for the persistent encapsulation of behaviors normally modeled by entire programs (which may be *fine-grain* object-oriented). CLOUDS objects send messages to each other via inter-object invocation. One CLOUDS object may invoke a public *operation* of another CLOUDS object, with or without parameters.

Threads are active entities which invoke public operations of CLOUDS objects. Threads, via object invocation, can execute code and manipulate data in multiple virtual address spaces. During each object invocation, a thread executes code and manipulates data within the invoked object. Parameters passed in an object invocation and returned at that invocation's termination are data, not addresses, since addresses in the context of one CLOUDS object are meaningless in the context of another. Multiple threads executing in the same object share the contents of that object's virtual address space. Of course, visibility of addressable data and code is limited to the current address space.

3.2 Distributed Shared Memory

The Clouds distributed system is comprised of network connected data servers, computation servers, and user workstations. In order to distribute Clouds objects and threads over these machines, Clouds provides user transparent support for distributing computation and storage.

Distributed shared memory (DSM) supports the notion of shared memory on a non-shared distributed memory architecture and enforces a coherence protocol among the data servers that preserves single-copy semantics for all objects [17]. Through DSM, the contents of any Clouds object may be accessed from any computation node. Further, an object may be shared among many computation nodes with the DSM system maintaining the consistency of its contents [10]. Clouds assigns a computation node to a computation thread based on system load or user-preference, in cases of user-directed load balancing. If both objects reside at the same computation node, the Clouds invocation is *local*, otherwise it is *remote*.

Once thread execution commences, the required object is brought in to the selected compute server via demand paging. In addition, synchronization support allows threads to synchronize their actions regardless of where they execute.

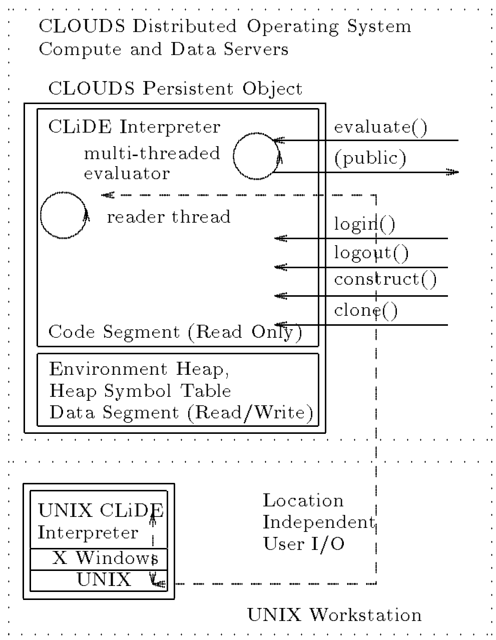

Figure 1: CLiDE Environment Structure

4 CLiDE Design

Clouds LISP Distributed Environments (CLiDE) is a distributed, persistent object-based, symbolic programming environment built on top of the Clouds distributed operating system. The features of Clouds provide us a substrate on which we can build a system that exhibits the characteristics presented in Section 2.1. Further, the ability to realize these characteristics at the operating system level validates the utility of the persistent, large-grained, object/thread computing system model.

Clouds objects are the repositories for persistent CLiDE symbolic processing environments. The Clouds invocation mechanism provides a means for sharing environments. Clouds objects with object-like interfaces allow construction of encapsulated, protected CLiDE environments. These interfaces, along with Clouds consistency mechanisms, provide a framework for maintenance of environmental consistency and security. Computation and storage distribution in Clouds supports construction of fault-tolerant decision support systems. And, as we shall see later (Section 6), Clouds large-grained persistent objects, coupled with CLiDE environments, support the architecture of future megaprogrammed information systems.

4.1 Environment Storage Model

A typical CLiDE system is composed of a group of cooperating Clouds persistent objects of the CLiDE class. Each of these persistent objects emulates a single-user, symbolic processing environment contained in a distinct, persistent, virtual memory address space. We call an individual symbolic processing environment a CLiDE *environment.* Multiple users can create, maintain, and modify their own user environments without the explicit loading and saving of environmental data because unlike typical symbolic processing environments, CLiDE environments persist until explicitly deleted.

4.2 CLiDE Environment Public Operations

At the core of a CLiDE environment (Figure 1) is a LISP interpreter, whose dialect is based partly on Common LISP [23] and *Kernel* [15], a portable LISP interpreter.

The functionality of a CLiDE environment is made available to users, other CLiDE environments, and Clouds objects representing classes programmed in CC++ (Clouds C++) or Distributed Eiffel [10] through an object-like interface that has five publicly available operations: `construct`, `clone`, `evaluate`, `login`, and `logout`.

The `construct` operation creates a new instance of a persistent CLiDE environment and assigns it a unique name.

The `clone` operation creates a copy of a current CLiDE environment instance by first instantiating a new CLiDE environment and copying all the CLiDE environment information from the cloned environment to the new one. Different types of CLiDE environments with certain characteristics, such as a Common Lisp Object System (CLOS) implementation and predefined CLOS object hierarchy, could be specified and cloned enabling the construction of a hierarchy of CLiDE environment sub-classes.

The `evaluate` operation enables other Clouds objects to perform LISP evaluations in a CLIDE environment. This extremely powerful operation is the conduit for environment sharing among a group of CLIDE environments. The CLIDE evaluation model is discussed in Section 5.2.

The `login` operation allows a user to attach a CLIDE environment to an *X Windows* server window. If the attach is successful, a *reader* thread is spawned which will read input directly from the user interface window and evaluate it in the CLIDE environment attached to it.

The `logout` operation allows the system operator to detach an X window from a CLIDE environment in case of a system mishap.

4.3 The CLIDE Interpreter

The interpreter in a CLIDE environment has a **multi-threaded evaluator**. The evaluator of CLIDE environment $\mathcal{A}$ is responsible for evaluating LISP expressions in the context of environment $\mathcal{A}$. Any portions of a LISP expression that should be evaluated in the context of another CLIDE environment $\mathcal{B}$ are sent to its multi-threaded evaluator via a CLOUDS invocation. The public `evaluate` operation of $\mathcal{B}$ is responsible for receiving requests for evaluation from other CLIDE environments, validating evaluation permissions, evaluating the expression, and finally returning the results.

A CLIDE environment **reader** thread is spawned when a user invokes an environment's `login` operation. The environment reader passes along LISP expressions to the evaluator for interpretation.

Since CLIDE environments persist, reclamation of space taken up by unreferenced heap objects is of primary importance. Our system design calls for a two-fold, **garbage collection strategy** similar to Wilson's Opportunistic Garbage Collector [27]. While evaluations are active within a CLIDE environment, generational garbage collection takes place when a user is not likely to notice. Further, as CLIDE remote evaluations complete, certain dereferenced symbols pertaining to it are automatically collected. When no evaluations are active in a CLIDE environment, the state of the environment heap is checkpointed and the tenured object generation[1] is garbage collected. If a remote evaluation is received by the CLIDE environment during this potentially lengthy garbage collection operation, the collection operation is aborted, and the environment state is rolled back. Attempts to invoke a CLIDE environment's `login` entry point during a tenured generation garbage collection operation are blocked until the operation completes. Reader and remote evaluation threads of control are used to spawn garbage collection threads. Utilization of object/thread migration mechanisms in CLOUDS allows garbage collection operations to be relegated to specific administrative sites in order to prevent performance degradation at compute servers involved in ongoing evaluation activity.

[1]Research has shown that there is a high mortality rate among recently created objects. A generational garbage collector scavenges this young group of objects most often. Objects that survive several garbage collection scavenges are considered "long-lived" and are *tenured* to a memory space that is garbage collected less often.

5 Using CLIDE

As previously noted, the functionality of a CLIDE environment is made available through an object-like interface utilized by specific users, other CLIDE environments, and CLOUDS objects of other classes. In the subsequent sections, we discuss the CLIDE user interface, the CLIDE evaluation model, and CLIDE environment security and consistency issues. Following this discussion, we give an example.

5.1 User Interface

The system configuration for CLIDE is a hybrid one consisting of multiple machines, each running either UNIX[2] or the CLOUDS distributed operating system. The CLOUDS and UNIX machines cooperatively interact to provide users with the benefits of both operating systems.

A UNIX-based CLIDE interpreter serves as the user interface conduit to CLOUDS-based persistent CLIDE environments. With this user interface setup, users can enter, test, and debug CLIDE functions in the UNIX-based CLIDE environment before moving them to a CLIDE environment running under CLOUDS. In addition, the CLOUDS operating system interface, instead of being a collection of cryptic "guru-mnemonic" commands (like UNIX), is a set of CLIDE commands which can be quickly assimilated by any LISP programmer. CLIDE persistent environments, combined with a UNIX-based CLIDE interpreter user interface strategy, allows the construction of a multi-user symbolic processing environment on a UNIX-CLOUDS system platform.

5.2 Evaluation Model

The CLOUDS invocation model supports CLIDE environment sharing through inter-environment *synchronous* or *asynchronous* evaluations. In order to see the effects of the CLIDE evaluation model on a group CLIDE environments, we will look at local, remote, synchronous and asynchronous evaluations.

5.2.1 Synchronous Remote Evaluations

In general, CLIDE evaluations are assumed to be *local* and *synchronous* unless explicitly specified otherwise. The CLIDE `remote` function is used to specify remote environment evaluations. The syntax of this LISP form is:

`(remote` <*remote-environment*> `<expr>)`

[2]UNIX is a trademark of UNIX System Laboratories, Inc.

For example, the LISP expression evaluated in CLIDE environment env_0:

```
(remote env1
   (lookup-function
      (remote env2 data-item)))
```

implies two synchronous, remote evaluations. First, an object invocation is made to the `evaluate` operation of CLIDE environment env_1. The expression passed to env_1 is:

```
(lookup-function
   (remote env2 data-item))
```

Now, the CLIDE environment env_1 makes an object invocation to the `evaluate` operation of CLIDE environment env_2. Thus, the expression `data-item` is passed to env_2.

CLIDE environment env_2 evaluates `data-item` and returns the results of this evaluation to env_1 which subsequently uses these results as an argument to its function `lookup-function`. Upon completion of this evaluation, the final results are returned to the original caller.

5.2.2 Asynchronous Evaluations

Parallel processing research in the late 70's introduced the notion of using asynchronous evaluations to exploit parallelism [16]. Later, the semantic implications of *futures* in LISP-like systems were addressed in Multilisp [14]. Since then, the *future* abstraction has been extended by others in order to ease programming of distributed applications with asynchronous execution entities. Examples include Argus [18] and Cronus [25]. The *future* abstraction in CLIDE is supported in part by CLOUDS, which provides asynchronous invocation and claiming of single threads, and a suite of CLIDE functions which take care of the identification, monitoring, and grouping of sets of asynchronous evaluations. Representative functions of this suite include: `create-future-set`, `clouds-invoke`, `claim`, `claim-status`, and `discard`.

The `create-future-set` function creates a *future set* structure and names it. *Future sets* group together one or more asynchronous evaluations into one abstraction that can be referred to at a future time.

Asynchronous object evaluations in CLIDE are triggered by the appearance of an *asynchronous evaluation directive* ('!') followed by the set name of a *future set*.

`!`<*set-name*>`( [remote `<*remote-env*>`]`<`expr`>` ... )`

CLIDE evaluations of this form return a positive integer identifying the evaluation within the future set. *Nil* is returned if the supplied *future set* structure does not exist.

Evaluations may be directed to a non-CLIDE Clouds object via the `clouds-invoke` function. This function generates a CLOUDS object invocation. Non-optional parameters include an *expression* to be evaluated, the *object name* of a CLOUDS object of any class, and the *operation* of the CLOUDS object to invoke. The *expression* is passed as a string to the receiving object. CLOUDS objects implemented in either DISTRIBUTED EIFFEL or CC++ must provide buffer space to capture and parse the contents of the string passed to it. The development of systems with modules that are programmed in different languages is facilitated by the ability of CLOUDS objects to invoke each other through *standard interfaces.* Thus, a system which has a calculation and real-time control module, a rule-base module, and a storage module based on fine-grained objects could be implemented in CC++, CLIDE, and DISTRIBUTED EIFFEL respectively, as a set of three persistent CLOUDS objects.

The `claim` function allows the user to claim results of one or more *future sets.* This function blocks (waits) until each *return expression* specified is true. The *return expression* is a logic expression that specifies which (if any) of a *future set's* evaluations must complete for the *future set* to be considered *claimable.* An example showing the construction of the *return expression* is shown in Section 5.5.

The `claim-status` function enables the user to check the completion status of one or more *future sets.* The function returns a list of lists, each containing the requested set name and the identifiers of the evaluations in that set that have completed.

Care must be taken to prevent the accumulation of stale or orphaned CLIDE future set structures since they are stored in the environment heap. Hence, the `discard` function, which is used to discard individual evaluation results or pending request(s) for results of a particular evaluation from a named future set. In addition, the discard function can remove the entire *future set* from the heap.

5.3 Environmental Security

Any system which provides support for a set of shared objects must provide a means to guarantee their consistency and integrity.

Since the contents of CLIDE environments are easily shared via CLIDE remote evaluation, the design of environmental security measures which guarantee CLIDE environment integrity are particularly important.

The CLOUDS distributed operating system provides security for CLOUDS objects by treating them as protected distinct virtual spaces. CLOUDS objects are semantically similar to fine-grained objects supported by object-oriented languages like Eiffel or C++ and may be accessed only through programmer defined object-like interfaces. Users must go through the CLOUDS operating system via these object-interfaces in order to access the contents of a particular CLOUDS object. CLOUDS objects are thus protected the same way as user spaces are protected from the effects of other users in conventional operating systems.

Although a CLOUDS object is protected, the contents of a CLIDE environment, encapsulated in a CLOUDS object, is still vulnerable due CLIDE's Lisp-like nature. Since remote evaluation within an environment is possible, that environment's contents can be modified maliciously unless additional security measures are taken. Each symbol in a CLiDE environment has an *access list* which states the permissions of access for that symbol. The access list for a symbol contains *access rights* for *local* and *remote* evaluations.

Through symbol level security, CLiDE environments with *object-based interfaces* are able to be constructed. All symbols in a CLiDE environment are assumed to be *private*. A programmer can define CLiDE functions within an environment that will serve as that environment's public interface and make their access level *public*. Thus, a CLiDE environment can encapsulate "code" and data as effectively as any Eiffel or C++ fine-grained object.

5.4 Environment Consistency

CLiDE environment consistency is provided through Clouds' *invocation-based consistency control* mechanisms [8]. Each public operation of a Clouds object has a *consistency label* which states the consistency requirements for the program code accessible from that entry point. This approach defines three consistency classes, *global*, *local*, and *standard*. Properly used, *global consistency* (GCP) guarantees that Clouds will automatically control consistency and recovery across a *group* of Clouds objects of any class. *Local consistency* (LCP), on the other hand, is used when the desire to maintain consistency is localized to one particular object. This consistency class guarantees that the system will control consistency and recovery so that one particular Clouds object stays internally consistent. *Standard* consistency (SCP) guarantees nothing; if a particular execution suffers no failures, the object data will be consistent, otherwise consistency is not guaranteed. The Lisp-like nature of the CLiDE environment requires that these consistency labels be extended to apply to any CLiDE symbol.

5.5 An Example

The following example demonstrates the power of the CLiDE evaluation model and system structuring paradigm.

Suppose we have implemented the following agents for the navigation system of an autonomous mobile robot. Each agent is encapsulated in a Clouds persistent object. Figure 2 shows the interrelationships between each of the agents.

Navigator. This agent is responsible for supervising the navigation of the robot. It is a CLiDE environment.

Cartographer. This agent collects sensory data from a high-speed vision system, makes inferences about the sensory data, and updates the world-map of the robot. It is a Clouds CC++ object.

World-Map. This CLiDE environment contains the robot's world map. This object is shared by the navigator and cartographer agents.

A portion of the navigator agent's code deals with answering the question "Where am I in the world?". In order to

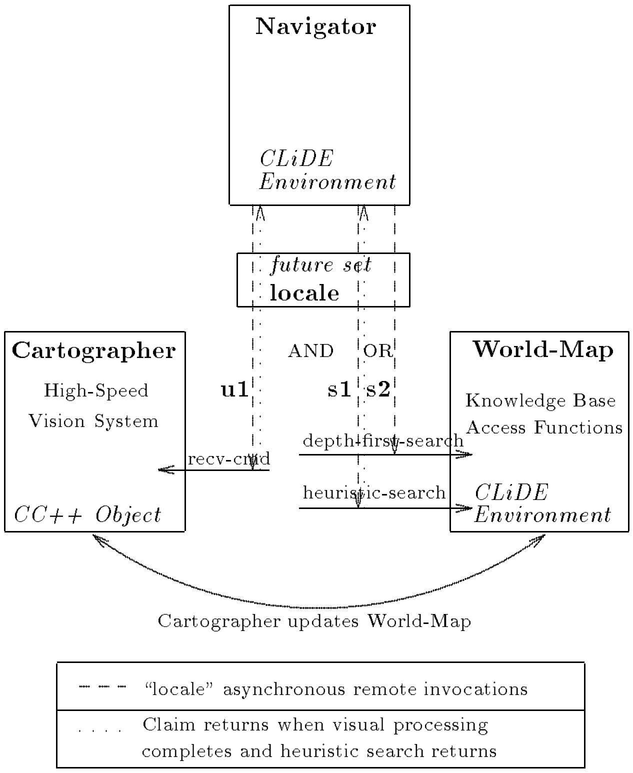

Figure 2: Robot Navigation System Agents

answer this question, the navigator must *update its world-map* and *find out where it is* in that map.

The code fragment that accomplishes these tasks looks like this:

```
1>> [GCP (prog (locale s1 s2 u1)
2>>  (create-future-set locale)
3>>  (SCP setq s1
      !locale(remote world-map
              (heuristic-search kb)))
4>>  (SCP setq s2
      !locale(remote world-map
      (depth-first-search kb)))
5>>  (setq u1
      !locale(clouds-invoke
      "do-visual-processing"
      "cartographer" "recv-cmd"))
6>>  (claim (locale '(and (or s1 s2) u1)))
7>>  (clouds-invoke
      "update-world-map"
      "cartographer" "recv-cmd") ) ]
```

First, the navigator begins a *globally consistent* program evaluation (line 1). Within this GCP evaluation, the navigator agent creates a *future set* named `locale` (line 2). The world-map CLiDE environment, via CLiDE security mechanisms, provides two publicly available methods for searching its knowledge base. The first method is a `heuristic-search` that works very well at times (depending on the data) but can run forever at other times (line 3). The second method is a `depth-first-search` that runs very predictably. When the `heuristic-search` runs well, it can be many times faster than the standard `depth-first-search` (line 4). So, the navigator makes two asynchronous, remote evaluations to the world-map CLIDE environment, which effectively begins the *simultaneous* search of the world-map knowledge base, `kb`, using the two different public methods. Note that both of these searches are conducted with *standard consistency* evaluations. This is *allowable* because the searches are read only and *necessary* because a GCP evaluation would not allow simultaneous searches in a CLIDE environment (world-map) it was enforcing consistency upon [7].

In addition, the navigator must update the world-map. Only the cartographer knows how to do this, so the navigator makes a final asynchronous, remote evaluation to the cartographer object (line 5). Now, the navigator waits until the *visual processing* (line 5) and either search (line 3 and 4) completes. When this condition becomes true, the `claim` call unblocks (line 6). The cartographer is then directed by the navigator to update the world-map with its newly gathered information (line 7).

The GCP evaluation makes sure that all three CLIDE environments stay consistent with respect to each other. If the GCP evaluation commits, then the changes made in the navigator, cartographer, and world-map environments will be flushed to stable storage and will be visible to subsequent evaluation threads. If the GCP evaluation aborts, the state of these environments would be the same as they were just prior to the initiation of the GCP evaluation, as if the GCP evaluation had never occurred.

6 The Megaprogramming Paradigm

Programming in the very large, or *megaprogramming*, deals with the development, construction, and maintenance of very large applications made up of large programs or *megamodules* with object-like interfaces [26]. A megaprogram is composed of a group of these modules which interact with each other through well-defined object-like interfaces. Generally, constructing the object-like interfaces for these *megamodules* requires programming semantics support beyond that of typical fine-grained object systems and support for these semantics must be provided by the operating system on which the megaprogram resides. This programming paradigm is particularly well suited for object/thread distributed computing systems like CLOUDS since the components of megaprogram construction and execution are the operating system's primary units of distribution.

CLOUDS simplifies large-grained object semantics somewhat by allowing programming to proceed as if all CLOUDS objects reside on one huge computer system. There is no need to support object location programming semantics for CLOUDS user object specification because, in most cases, location and migration issues are taken care of automatically. Location specific objects which perform particular tasks may be anchored to a particular hardware platform, but as far as the user is concerned, that location is one, global machine. Thus, all CLOUDS objects are referenced in the same manner, whether they are physically distributed or not. As shown in Section 5.5, CLIDE environments which encapsulate particular behaviors and knowledge representations can be constructed. Through CLIDE's straightforward evaluation semantics, CLIDE environments can be programmed to cooperate among themselves to find the solution to a complex problem. Further, the integrity and consistency of these intelligent agents is maintained through CLOUDS system consistency mechanisms and CLIDE security mechanisms.

Persistent environments in CLIDE are an appropriate storage medium for the symbolic processing megamodules of a megaprogram. Transparent storage and computation distribution in CLOUDS allows thread-based megaprograms to run on loosely coupled, distributed hardware. In addition, megamodules can be specified in the language most appropriate for the task. The CLIDE system on CLOUDS provides DAI researchers a means to physically distribute DAI software architectures and integrate them with heterogeneous megamodules based on different non-symbolic software paradigms to create distributed artificial intelligence megaprograms made up of CLIDE environments and CLOUDS objects programmed in languages like CC++ and DISTRIBUTED EIFFEL.

7 Implementation and Future Work

Implementation of a single threaded version of CLIDE for UNIX (in C++) and CLOUDS (in CC++ [10]) is complete. Current implementation work is focused on multi-threading the CLIDE environment. This task has required design work on an evaluator that can handle multiple evaluation contexts while sending and receiving native heap data to and from other CLIDE environments. In addition, design work has begun on truly encapsulated fine-grained intra-heap objects which will be used to simplify the specification of inter-environment evaluation security, and provide functionality similar to Common Lisp packages and CLOS objects.

An implementation of CLOUDS consistency mechanisms which supports *consistency labeling* of CLOUDS object entry points is complete [7]. An intra-environment version of these mechanisms will need to be implemented in order to support consistency for individual CLIDE evaluations and symbols.

References

[1] N. Aiello. User Directed Control of Parallelism; The CAGE System. In *Proceedings of the Expert Systems Workshop*, pages 146–151, Pacific Grove, CA., April 1986. Asilomar.

[2] R.C. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112, August 1989.

[3] John K. Bennett. The Design and Implementation of Distributed Smalltalk. *SIGPLAN Notices*, 22(12):318–330, December 1987.

[4] John K. Bennett. Experience With Distributed Smalltalk. *Software - Practice and Experience*, 20(2):157–180, February 1990.

[5] D. Bobrow, L. DeMichiel, R. Gabriel, G. Kiczales, D. Moon, and S. Keene. The Common Lisp Object System specification. Technical Report TR 88-002R, X3J13 standards committee document, 1988.

[6] D.A. Carlson and S. Ram. An Object-Oriented Design for Distributed Knowledge-Based Systems. *Proceedings of the Twenty-Second Annual Hawaii International Conference on Systems Sciences*, III:55–63, January 1989.

[7] Ray Chen and Partha Dasgupta. Implementing Consistency Control Mechanisms in the Clouds Distributed Operating System. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, May 1991.

[8] 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.

[9] S. Clamen, L. D. Leibengood, S. Nettles, and J. Wing. Reliable Distributed Computing with Avalon/Common Lisp. In *1990 International Conference on Computer Languages*, pages 169–79, 1990.

[10] P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra, M. Pearson, R. Chen, and C. Wilkenloh. Language and Operating System Support for Distributed Programming in Clouds. In *Proceedings of the 2nd Symposium on Experiences with Distributed and Multiprocessor Systems*, 1991.

[11] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, et al. The Design and Implementation of the *Clouds* Distributed Operating System. *Usenix Computing Systems*, 3(1), 1990.

[12] L. Gasser, C. Braganza, and N. Herman. Implementing Distributed AI Systems Using MACE. *Proceedings of the Third Conference on Artificial Intelligence Applications*, pages 315–320, 1987.

[13] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Menlo Park, CA., 1983.

[14] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[15] Sharam Hekmatpour. *LISP: A Portable Implementation.* Prentice Hall International (UK) Ltd., Hemel Hempstead, United Kingdom, 1989.

[16] Henry Baker Jr. and Carl Hewitt. The Incremental Garbage Collection of Processes. *SIGPLAN Notices*, 12(8):55–59, August 1977. Proceedings of the Symposium on AI and Programming Languages.

[17] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[18] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *SIGPLAN Notices*, pages 260–267, Vol 23, No. 7, July 1988.

[19] D.J. MacIntosh and S.E. Conry. A Distributed Development Environment for Distributed Expert Systems. *Third Annual Expert Systems in Government Conference*, pages 72–79, 1987.

[20] Andreas Paepcke. PCLOS: A Critical Review. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1989.

[21] J. Rice. Poligon, A System for Parallel Problem Solving. In *Proceedings of the Expert Systems Workshop*, pages 152–159, Pacific Grove, CA., April 1986. Asilomar.

[22] J. Stamos and D. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–65, October 1990.

[23] G. Steele. *Common LISP*. Digital Press, Burlington, MA., 1984.

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

[25] R. Walker, E. Floyd and P. Neves. Asychronous Remote Operation Execution in Distributed Systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 253–259, May 1990.

[26] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):8–85, August 1990.

[27] Paul Wilson and Thomas Moher. Design of the Opportunistic Garbage Collector. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1989.