NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

Parallel Processing with Agora

Alessandro Forin, Roberto Bisiani, and Franco Correrini

December 1987
CMU-CS-87-183

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Copyright © 1986, 1987, 1988 A. Forin, R. Bisiani, F. Correrini

This research is sponsored by the Defense Advanced Research Projects Agency, DoD, through ARPA Order 5167, and monitored by the Space and Naval Warfare Systems Command under contract N00039-85-C-0163. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the United States Government.

Table of Contents

Abstract
1. Introduction 1
 1.1. Why Agora ? 3
 1.2. The User's View 3
 1.3. Leveraging on Agora 5
2. The Compiler 6
 2.1. The Language 9
  2.1.1. Data Definitions 9
  2.1.2. Addressing 9
  2.1.3. Access Function Definitions 10
 2.2. Translation 11
 2.3. The Output 12
 2.4. Examples 13
3. Agora From C and C++ 14
 3.1. C Agents 19
 3.2. The ASM library 19
4. Agora From Lisp 21
 4.1. Lisp and Alien Languages 25
  4.1.1. The Agora-Lisp Shell 25
  4.1.2. Lisp Agents 25
 4.2. Notes 26
5. The Shell 26
 5.1. Extending the Unix Environment 31
 5.2. The AgoraShell 31
  5.2.1. Agent Control 32
  5.2.2. Data Manipulation 33
  5.2.3. Miscellaneous 35
 5.3. Living in Agora 35
6. The Networked Shared Memory 36
 6.1. Memory and Messages 37
 6.2. The AgoraServer 37
 6.3. Notes 38
7. Agora-Extras 40
 7.1. CFrames 45
  7.1.1. Language Overview 45
 7.2. Xlisp 46
8. Other Tools 50
 8.1. Garbage Collection 53
 8.2. Performance Monitoring 53
 8.3. Pattern Matching 54
 8.4. Debugging 56
 8.5. Data Storage 58
 8.6. Loading of Agent Images 60
60

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

9. Example
9.1. A Name Service
9.2. A Distributed Make
10. Conclusions
References
Index

Abstract

Agora *n*. 1. *Greek* The public square and marketplace in an ancient Greek city; 2. *Comp*. An environment for the construction of large scale parallel and distributed systems, which supports multiple languages and heterogeneous computer architectures.

This document is a detailed report on the current status of the **Agora Project**. Its aim is to describe the current design and implementation as well as the immediate goals and future research directions of the project. Its intended audience includes both fellow researchers and potential users of the system. The material contained in this document is detailed enough to constitute a practical guide to parallel programming with Agora.

Chapter 1 is an overview of the project's key ideas: mildly curious readers may wish to concentrate on this chapter. Chapter 2 describes Agora's intended approach to programming parallel applications. Both interested readers and potential users should look into it. Chapters 5, 3 and 4 describe in detail the core components of the system, and Chapters 6, 7 and 8 describe the remaining components. These are reserved for potential and actual users, although each Chapter typically contains an introductory section that is of more general interest. Chapter 9 is a complete example. Most readers will probably want to look at it, at least superficially.

Acknowledgements

Other past or present members of the Agora Project include, in alphabetical order, F. Alleva, V. Ambriola, M. Bauer, A. Brown, E. Hughes, F. Lecouat, R. Lerner, D. Tomm: their contributions have been invaluable in making Agora a reality. Special thanks are due to Duane Adams for his continuous help and his precious comments while revising our documents.

2

1. Introduction

This chapter provides an overview of the Agora Project: it illustrates the basic motivations behind it, and the key ideas that are being explored. Specific sections contain: a description of the role that Agora plays for programming parallel applications on heterogeneous architectures and languages, a description of a specific scenario for parallel programming, and a brief mention of research areas that can profitably leverage on the results of the project. Chapter 10 summarizes the most novel features that Agora (as a programming system) provides.

In this document we have attempted to be as complete as possible, previous documents [6, 7] contain our initial ideas. The final word on implementation details is left to the Agora User Manual [21].

1.1. Why Agora ?

The solutions of many real-life problems encountered in science and industry require the integration of parallel programs written in different languages and running on heterogeneous machines. We call the development of such systems *heterogeneous parallel programming*. For example, sensor data acquisition and signal processing might have to be integrated with planning, or electrical circuit simulation might have to be integrated with expert system technology. The aim of the Agora project is to facilitate heterogeneous parallel programming.

As a possible scenario, imagine a multi-media dialogue system like the one that the MINDS project [43] is building at CMU. In this case, a dialogue manager, written in Lisp and running on a workstation, uses as one of the input devices a parallel speech recognition subsystem largely written in C and running on a shared memory multiprocessor. The speech recognition subsystem is the result of a separate research project, but it must be tightly integrated with the rest of the system. Other input devices allow textual and graphic input, and they run on another workstation. Another component of the system is a moderately sized database, which is kept on a separate workstation and managed by a separate subsystem. Applications like this one use heterogeneous architectures, including multiprocessors (in the example Vax, IBM-RT and Encore Multimax), heterogeneous languages (C and Lisp), and need the integration of large and separately developed subsystems.

Modern, general purpose operating systems like Mach [4] offer interprocess communication facilities based on exchange of messages, sharing of memory, or both. There is a growing body of tools to deal with message based interaction, and in large part these tools rely on the Remote Procedure Call (RPC) paradigm. Message primitives, however, are bound to be relatively expensive: if correctly implemented they require the protection that only operating system calls can provide. Shared memory can provide higher efficiency with lower protection, but no existing system supports shared-memory-based interaction between heterogeneous languages. Agora covers exactly this missing part, offering enough tools and functionality to easily develop shared memory based system with heterogeneous components. Agora also extends the notion of heterogeneous shared memory across a network of workstations, in a totally transparent way. Agora can also be extended to ensemble machines like the Cosmic Cube [36], but we have not done it yet.

The most common tool for message-based interprocess communication is a *Stub Generator*, e.g. a

compiler that takes the specification of a procedure as input, and generates code to pack/unpack arguments in messages, send and receive them, and even structure the user's code in a *client-server* scheme. After learning the use of such tool, a user can concentrate on an almost-sequential model of parallel computing: clients invoke remote operations just like standard procedures, and writing a server largely consists of writing those procedures. Therefore, the low learning cost is one of the most appreciated characteristic of the Remote Procedure Call paradigm. Unfortunately, the notion of a client-server relationship is easy to understand, but not general enough. A server can provide fair allocation of common resources and a simple and safe interface to access them. Therefore, a server can be very useful in implementing operating system functions but is not suited to other problem domains, e.g. symbolic computing.

Agora supports the **Shared Data Type** paradigm: processes (agents, in Agora terms) communicate via controlled accesses to common data structures, rather then sending information inside messages. Sharing in Agora goes beyond the boundaries of physical shared memory: the same data structure is accessible and kept consistent across processors, and converted, if necessary, between different machine architectures (e.g. Sun versus Vax architectures). Data structures are strongly typed: each has a runtime type descriptor associated with it. Data structures are only accessed through user defined functions. Data structures can be created dynamically, and their type must be indicated at creation time. The act of modifying a data structure is an event that can be reported to all interested agents. This makes a shared data structure behave as an *active object*, since touching it constitutes an interprocess communication event. In this sense, execution in Agora is event-driven.

The notion of Shared Data Types borrows from Abstract Data Types [31] and Object Oriented Programming [26, 11]. In the sequential world, these techniques have gone a step beyond plain procedural decomposition for code design and structuring. In particular, object oriented languages favor the quick development of large evolutionary systems. We exploit these techniques in Agora because we expect them to be even more effective for the parallel world: not for hiding the parallelism, but for exposing and controlling it in a natural way. The notion of Monitors [29] is also related: procedures inside a monitor are the only ones allowed to operate on the monitor's variables. The basic control scheme that Agora exploits has also roots in BlackBoard Systems [17] and other AI environments [10].

In the same way a Stub Generator is a fundamental tool for defining (remote) procedural interfaces, a similar tool plays a central role in defining shared data types. We call this tool **Data Type Generator** because its purpose is to generate data type declarations to match the shared data structures, and appropriate functions to access them. This is done by compiling a description given in a simple meta-language into each of the supported programming languages. Contrary to Stub Generators, the Agora compiler also allows the definition of access functions (e.g. operators, methods) in the meta-language. The Agora meta-language is a simple subset of Lisp, but the specific language syntax chosen should not matter: only control constructs that are compatible across many languages are used.

The choice of living in an **heterogeneous environment** has also intrinsic advantages. With Agora, components can be coded in different languages, or new components incorporated in an existing system without extra effort. Subsystems can be tailored to the languages and/or architectures that are most suited to their efficient execution. Having a larger set of choices for the implementation encourages

devising more portable algorithms.

1.2. The User's View

Parallel systems in Agora are designed around the definition of the shared data they use. This definition is precise, unambiguous, and in form of a machine readable specification. The specification is then refined into executable code. The specification language used has a Lisp-like syntax. The definition of access functions follows the syntax of CommonLisp's *defun,* and includes forms like let, do, cond, etc. The language and its compiler are described in detail in Chapter 2.

Agora is implemented as a collection of programs and libraries, with specific versions for the various programming languages and machines. The core system includes:

- the AgoraShell, a modified version of the Unix C-shell;
- the AgoraCompiler;
- a runtime library (and lisp package) that defines the basic Agora primitives;
- the AgoraServers that take care of the network protocols.

Other basic tools include

- the Agora debugger,
- the parallel and incremental garbage collector,
- the performance monitor,
- a utility to save/restore on disk files the content of the runtime data-structures.

Programmers and users interact with Agora through a shell that could either be the AgoraShell or the Lisp interpreter. Both offer interpretive access to the same set of Agora primitives, and they offer further functionality to inspect and monitor the status of the shared memory. It is worth noting that Agora extends the basic capabilities without sacrificing any of the features that either the C-shell or Lisp offer. In this way the transition to a parallel environment becomes as easy as possible. Standard Unix/Lisp tools are used for program composition: editors, compilers, linkers etc. A C-program written in Agora (an Agora C-agent) looks to the AgoraShell exactly like any other Unix program. Therefore, Agora can also be used to develop parallel programming tools (e.g. a parallel-make utility that can spread compilations over more than one machine).

A shared data structure definition is nicknamed *context*. Agora provides a runtime library of standard functions to create, destroy, read and write contexts as procedural extensions of each of the supported languages. Users also define other *custom access functions* to provide operations that are more suited to manipulate the specific abstractions they use. These functions are implemented using the standard library functions.

The description in the meta language is handed to the AgoraCompiler which generates runtime type descriptors and language specific code. The code is then used in the various agents that make up the final application. Each agent can be tested in isolation, and to this end the shell (under the user's control) can provide all the stimuli that the other agents generate in the complete application. The shell is an interpreter (of either Lisp or C), and therefore simple scripts/functions are easily created and thrown

away after use.

Systems can be developed on a single machine without any concern of the final assignment of agents to processors in a distributed application. AgoraServers are the only components that are concerned with extending the notion of a shared memory across separated processors. The only Agora primitive that deals with the notion of separate processors is the one that starts a new agent, which optionally takes a specific host name parameter.

Since Agora is used to build large distributed applications, the debugging phase becomes especially important to shorten the development times. Agora provides a distributed debugger that is suited to the Agora's requirements: it can cope with multiple languages through language-specific sub-tools which are familiar to the programmers (e.g. modified versions of C and Lisp debuggers), it works across multiple machines and has full control over the inner workings of the Agora shared memory, and finally promotes a very simple and effective debugging protocol. The user executes the application without the debugger and if-and-only-if something is wrong the debugger is invoked to *replay* the exact history of the execution.

To allow faster access, data memory is largely managed in a write-once fashion in Agora. This makes it necessary to recover reusable storage to minimize virtual shared memory requirements, and to cope with limited addressing that certain architectures provide. Agora provides a garbage collector agent, startable at any time, which incrementally and in parallel reclaims reusable storage. The user has full control of the agent: in this way the garbage collection process will not interfere with time critical applications by executing at inappropriate times.

In order to run experiments and exchange data, it must be possible to save and restore the runtime data structures on disk files. This is done by invoking a simple agent that is also part of the Agora toolset. The representation on files is machine independent, so that heterogeneous machines can access the same data files, avoiding the waste of duplicating and converting them. Agora uses a binary representation because using an ASCII representation is both slow and wasteful. The information about the type of the data structure is used to convert it, when necessary, across incompatible architectures.

The following chapters provide more details about the various components of the system, and appropriate material to provide the *programmer's view* of Agora. Chapter 9 contains example applications.

1.3. Leveraging on Agora

Parallel programming is still at its beginning, and we should expect programming environments for parallel applications to become a sensible area for research in the near future. Agora can be effectively used as the basis for building advanced programming environments that use techniques from AI and Software Engineering to simplify the management of large systems. To prove this, we have built prototypes of the key components that such environments will use ([7, 9]). This included a distributed database based on a distributed frame language derived from SRL [22], a framework for easily defining customized environments in an object oriented style (this included a language for defining database

methods, to be usable both from Lisp and C agents), a planner that uses the description of the system to accomplish the user's goals via automatic invocation of the tools of the environment, and a graphical user interface that allows the user to peruse the database in a friendly way.

Interest upon BlackBoard Systems is growing both in industry and academia. An important feature that is lacking in existing systems is true parallelism: they all simulate it via interleaved or sequential execution on a single processor. Agora has already the necessary basic components of a BlackBoard system: it is easy to extend it into this direction. What is missing is a more powerful database definition and browsing capability, like the one that GBB [15] offers.

The parallelism that currently Agora offers is not at the very fine grain: starting parallel computations, and synchronizing them is still somewhat expensive. Agora should be able to use light-weight processes to lower the cost of starting new agents. The only way to lower these costs is to use a parallel language and its compiler: Multilisp [27] and Modula [42] among the best candidates. These languages can be integrated in Agora just like any other sequential language: components written in these languages can interface to other components using the Agora primitives. Integrating a language that covers fine-grain parallelism would lead to a system that covers all the possible application requirements.

8

2. The Compiler

This Chapter describes a tool, the Data Type Generator, that plays a central role in Agora. The tool takes as input a specification of the Shared Data Structures and generates type descriptors for the Agora runtime system and data type and function definitions for the programming languages that Agora supports. The specification is given via a meta-language, described in Section 2.1. The language is a simple subset of Lisp that allows definition of the shared types and of the operations on instances of the types, e.g. Shared Data Type definitions. The tool itself is described in Section 2.2, and the output it generates is described in Section 2.3. Finally, section 2.4 contains some examples.

2.1. The Language

Parallel systems in Agora are designed around the definition of the shared data they use. This definition is in the form of a machine readable specification. The specification is refined into executable code, to be used by the programmers that code the various components (agents) of the application. The specification is the sole mean by which interactions among agents are explicitly defined. The specification language allows the definition of both data and access functions.

Shared data structures are nicknamed *contexts*. A description of a context is composed of two parts: data definitions and access-function definitions. Data definitions specify the representation of the abstract data type, e.g. the structure of the data elements. Access functions specify the permissible operations on instances of the type, and how they are implemented. Agora provides standard functions to create, destroy, read and write contexts as procedural extensions of each of the supported languages as part of its runtime system. However, the standard functions may not suffice to express the exact semantics of the type. The language can specify access functions that define operations more suited to manipulate instances of the shared type. These functions are then compiled into each of the programming languages.

Three kinds of definitions can be part of a compilation unit: data definitions, addressing definitions, and access function definitions. These are each described in turn in the following. The language also encourages the creation of modular specifications. Visibility is restricted to the single compilation unit (a file), and it is possible to reuse specifications via the **include** operator. Including a file simply makes all the definitions contained in it visible.

2.1.1. Data Definitions

Data elements are typed and are described by using a set of primitive types (listed in the following) and array and record constructors. A type can be used within other types, hierarchically. A type is introduced by the keyword **deftype** and the name of the type that it defines. This is followed by a list of fields. Each field declares its own type, name and, optionally, an array size. The type of a field can either be a basic type or another user type. Recursive or incomplete definitions are not permitted, e.g. a type cannot have a component of the type itself. There are no pointer types.

```
(deftype Point
  (int16 x)
  (int16 y)
  (int8  scores 10))
```

In this example, the type *Point* includes two 16-bit coordinates *x* and *y*, and an array of 10 8-bit *scores*.

Enumerated types specify the set of values for the type before the type name, e.g. as in

```
(enum (red green blue) color 10)
```

The set of basic types includes the following:

- **int8**: byte
- **int16**: signed small integer
- **int32**: signed integer
- **uint8**: unsigned byte
- **uint16**: unsigned small integer
- **uint32**: unsigned integer
- **float32**: single precision floating point
- **float64**: double precision floating point
- **enum32**: enumerated type

The suffix in a name indicates the size in bits of values of the type. Array can have a variable size: a size of zero indicates a variable sized array. There can be at most one variable sized component per data element and it must be the last one (an implementation restriction). Using variable sized types in other types makes them variable sized as well, and the same restriction applies.

2.1.2. Addressing

Data elements define the granularity of the structured shared memory. A context definition includes a number of these types, and the definition of how the context is addressed. Agora supports two forms of addressing: through indexing in a linear array and through hashing of strings in a hash table. The definition of the shared data type is introduced by the keyword **defcontext** and includes the name, the specification of the addressing mode, and an indication of the types of the data elements in it. The addressing specifier can be either **:linear** or **:hash**. This can be followed by an optional specification for the range of the index (for linear addressing) or the size of the hash table (for hashed addressing). The defaults are 0 through infinity and a small table size, respectively.

```
(defcontext Screen :linear (1 1024*1024)
  ((data-ref pixel a_point))     ;; list of references
  (data a_point Point))          ;; data elements
```

In this example, the context *Screen* accepts a linear index ranging from 1 to 1024*1024, and includes the data element *a_point* of the previously defined type *Point*. Applying an address to a context yields a list of references. In the example, the list contains only one reference, *pixel*. References could either be (pointers to) data elements or addresses for other contexts. References to data elements are indicated by the keyword **data-ref**, references to contexts by the keyword **context-ref**. In both cases the name of the reference is given first, followed by the name of the referred-to object. These objects must be specified separately to allow multiple references to the same data elements. Data elements are specified by the **data** keyword and other contexts by the **context** keyword. In both cases the name of the object is

followed by its type.

For convenience, the type of a referred-to context can be defined in-line so that its components can be referenced from the outside *and* viceversa. This allows the construction of arbitrarily complex data structures. For example

```
(defcontext ColorScreen ..
 (<definition of the references>)
 (context Red ..
  ((data-ref red_pixel  pixels)))
 (context Blue ..
  ((data-ref blue_pixel pixels)))
 (data pixels Point)
```

Note that the inner context definitions both refer to the same data elements, i.e. *pixels*.

2.1.3. Access Function Definitions

Agora provides a set of *basic* functions for the creation and manipulation of contexts. These are available in language-specific libraries or are synthesized by the compiler. They are listed in Table 2-1.

Function	Description
cx_create(*cx_type, init_info* **)**	Creates a new instance of a context of the given type. The optional parameter *init_info* specifies a list of existing contexts to be shared.
cx_destroy(*cx* **)**	Reclaims all the storage space of the named context and destroys it.
cx_atomic_execute(*cx, function_call* **)**	Guarantees that, during evaluation of the given *function call*, modifications of the given context are deferred. This is valuable in those cases where a large mutual exclusion grain is needed, e.g. across consecutive read/write operations.
cx_read(*cx, address* **)**	Dereferences the given *address* for the given context and returns either nil or the list of references, e.g. pointer(s) to element(s) and/or address(es) for other context(s).
cx_write(*cx, address, values* **)**	Stores a new data element(s) in the context and makes it(them) addressable through the given *address*. A variation of *cx_write* allows for checking the *values* to see if they exist in shared memory, thus saving storage in case of sparse structures.

Table 2-1: Context Access Functions

A user will often use only the set of functions of Table 2-1. These are automatically provided by the description of the data structure. In particular, *cx_create* is always synthesized by the compiler, and creates all the necessary maps used for address dereferencing. Sometimes, a user may wish to provide a better encapsulation of a data structure, by defining other access functions in the same language that is used to describe the data structures. Implementation of these customized operations is defined in terms of the functions of Table 2-1.

An access function definition is similar to a Lisp function definition. It is introduced by the keyword **defaccess**, followed by the name of the function, by the parameter list, and the function body. Note that the parameter list will always contain at least a formal for the context instance. A function definition can be introduced by the keyword *defun*, and in this case it is intended that the function will not be made visible to the user.

```
(defaccess <name> (<params>)
  <optional documentation>
  <code>)
```

It is a non-easy task to decide what to put in a language and what not to. In our case, things are a little easier with Lisp, but still one would like to support as many features as possible. We do not, however, want to make the task of supporting a new language exceedingly difficult or cumbersome. The constructs that *must* be supported are repetition, conditional, assignment, and variable binding. In CommonLisp these are expressible with the forms **do cond set let**, respectively. The most restrictive type of these forms is always preferred. The rationale behind this is that access functions are likely to be used if-and-only-if they are simple and generate efficient code.

It is conceivable that an access function may need to directly manipulate the maps that are used for addressing, although this has not been the case for the applications we have seen so far. For instance, a user may wish to implement addressing forms different from the standard one. The functions that manipulate the addressing maps are defined in Table 2-2.

---

map_update(*map, address, references* **)**
: **Updates the entry at** *address* **to refer to the given (list of)** *references*.

map_find(*map, address, allocate* **)**
: **Dereferences the given** *address* **and returns the corresponding entry. If the entry does not exist,** *allocate* **tells whether to create it or not.**

map_next(*map, address* **)**
: **Returns the first valid address that follows** *address***, if any.**

map_bounds(*map, first, last* **)**
: **Returns the lower and upper bound of a linear map.**

Table 2-2: Map Access Functions

2.2. Translation

The Agora compiler actually consists of two components: the **Agora Type Generator (atg)** and the **Agora Type Encoder (ate)**. The first one does most of the work, the second one simply takes a readable representation of the data structures produced by *atg* and packs it into a more compact form. *atg* has been produced with the Unix compiler-compiler *yacc*, and includes a scanner, a parser, semantic checker, and code generation. It contains multiple code generators, all of them working on the same parse tree. The syntax is so simple that it does not create any parsing or translation problem. The only exception is the translation of the Lisp code into C, which is done by a separate module.

atg takes as input a single description file and produces a number of output files. Agora currently supports C (and C++) and Lisp, so the output files are '.h' header files and '.c' code files for C/C++ and '.lisp' files for Lisp. A file with a '.dump' suffix contains the type descriptors needed at runtime. This last file is produced by *ate* and can be directly loaded in shared memory either by the shell or any other agent, using a standard library function.

2.3. The Output

Each Shared Data Type, described in the meta-language, is compiled into a number of representations. These include representations specific to the the languages supported by Agora and a representation that is used in the Agora runtime system (runtime type descriptors). These are briefly described in the following. Examples can be found at the end of this Chapter, and in Chapter 9.

C and C++

The Agora types and addressing definitions are translated into C structures defining C types with the same names and the same components. More precisely, each *deftype* produces a C structure definition for the data element, and each *defcontext* produces a C structure that correspond to the reference list.

The basic types are translated as follows

```
int8    ---> char
int16   ---> short
int32   ---> int
uint8   ---> unsigned char
uint16  ---> unsigned short
uint32  ---> unsigned int
float32 ---> float
float64 ---> double
enum32  ---> enum
```

Lisp

The Lisp code makes use of the CMU-CommonLisp foreign-function facility [32]. It follows rules similar to C: for each declaration *atg* generates a **def-c-record** call, which has effects similar to a *struct* definition in C. This is the lowest level at which Lisp can access shared memory, and clearly the most efficient. Direct use of the foreign-function facility is however quite user-unfriendly. The compiler therefore creates an extra layer that masks the facility and provides access directly in terms of Lisp native objects. At this level, each structure is defined as a *defstruct* type. Accessing a context is in terms of Lisp structures. A defstruct type defines the reference list and other defstruct types define the data elements.

The basic types are translated as follows

```
int8    ---> fixnum, string if array
int16   ---> fixnum
int32   ---> fix/bignum as appropriate
uint8   ---> fixnum
uint16  ---> fixnum
uint32  ---> fix/bignum as appropriate
float32 ---> float
float64 ---> float
enum32  ---> fixnum
```

Runtime System

The descriptors generated by *ate* for the shared data types are simple tables of field descriptors. Each field describes an array of objects Each object is either an array, or a structured object. Structured objects are expanded inline. Each descriptor has a tag, a field name, a field size, and a multi-purpose informational field. These descriptors serve three purposes in Agora:

- type checking;
- data transformation across heterogeneous machine architectures;
- save-restore of data structures between memory and permanent storage.

2.4. Examples

In this section we describe two simple examples taken from the literature. They illustrate how abstraction techniques are applied to parallel programming in Agora. The first example is the set of integers described in [31], extended to support parallel operations. The second example is a classical example for transaction processing: the example of a checking account. We took it from [2], and used it to stress some features of Agora.

Parallel Set of Integers

The following data structure implements an unbounded set of integers e.g. a set in the mathematical sense. As a data type, it supports operations like creation, insertion and deletions of a member, test for membership, iteration over the set. Besides those defined in [31], we have added an extra operation that is exported to the user: the test for emptiness of the set. Note also that we relax the restriction that *size* only be applied to a non-empty set: the size of an empty set will be zero.

```
;;
;; Context definition
;;
(defcontext parintset :linear (0 infinity)
  ((context-ref rep parintset)))

;;
;; Exported functions
;;
(defaccess create ()
  "Creates a new empty set of integers"
  (let ((pset (parintset-create)))
    (parintset-write pset 0 0)
    pset))

(defaccess size (pset)
  "Returns the number of members of the set pset"
  (parintset-rep (parintset-read pset 0)))

(defaccess emptyp (pset)
  "Tests if the set pset is empty"
  (= 0 (size pset)))

(defaccess insert (pset val)
  "Adds the integer val to the set pset"
  (parintset-with-locked-cx pset
    (seq-insert pset val)))
```

```
(defaccess delete (pset val)
  "Removes the integer val from the set pset"
  (parintset-with-locked-cx pset
    (seq-delete pset val)))

(defaccess member (pset val)
  "Tests if the integer val is a member of the set pset"
  (getindex pset val))

(defaccess choose (pset)
  "Returns an arbitrary member of the set pset"
  (parintset-read pset (random (size pset))))
;;
;; Non exported functions
;;
(defun seq-insert (pset val)
  (cond
    ((member pset val))
    (t (parintset-write pset
         (parintset-write pset 0 (+ 1 (size pset)))
        val))))

(defun seq-delete (pset val)
  (let ((idx (getindex pset val)))
    (cond
      (idx
        (parintset-write pset idx
          (parintset-rep (parintset-read pset (size pset))))
        (parintset-write pset 0 (- (size pset) 1))))))

(defun getindex (pset val)
  (let ((high (size pset)))
    (do ((i 1 1)
          (or
            (> i high)
            (= val (parintset-rep (parintset-read pset i))))))
    (cond
      ((> i high) nil)
      (t i))))
```

To implement the set we use a simple linear context, with a map that contains a single reference. Since references to linear contexts are integers, they can be used directly to represent the members of the set. All functions with the *parintset-* prefix are automatically defined when the context is declared and can be used in the definition of the access functions. In particular, *parintset-read* takes the context and an index as parameters and returns the corresponding entry in the map, while *parintset-write* takes the context, the index and the new value to be written in the context.

The access function *create* calls the standard context creation operation that has been defined for the type *parintset*, and then sets the first entry of the map to 0. We use this entry to hold the number of members of the set, which is what the *size* operation returns. Note the wrapping of *parintset-rep* around the reading of the context, to select the field *rep* of the map: in general maps can hold multiple fields, therefore the returned value is a record. The operation *emptyp* uses *size*, since we defined an empty set to

have size 0.

The insertion and deletion functions modify the set through a sequence of operations: the call to the sequential versions of the operations is therefore wrapped in a *with-locked-cx* form to guarantee mutual exclusion. This form is the only one that explicitly refers to mutual exclusion, all other basic functions manage locks internally but do not need to let the user know it. Locks are recursive, e.g. the same agent can call *with-locked-cx* within itself, but there is no magic against deadlock: shared data structures that provide operations with mutually recursive locks should be considered with suspicion, as they probably originate from a poor design. If two data structures are so closely related to require mutual locks, they are indeed better defined as a single, hierarchical data structure.

The *member* operation calls the internal function *getindex*, which returns the index of the member in the map. The *choose* access function is an *iterator* [31]. Iterators are very useful operations that provide a mean for e.g. iteration over the members of a composite data type. Like *choose*, iterators for shared data structures must avoid keeping state in the shared data structure itself, since the next call could be from a different agent. The correct scheme is to define iterators similar to the *successor* function for integers, e.g. functions without side-effects. The specifics of the iterator depend very much on the intended use, especially when performance is important (e.g. inner loops).

The non-parallel implementation of the insertion function uses the size indicator to put the new member at the highest index in the map, unless the integer is already part of the set. Removing the membership test would lead to a different kind of mathematical object: a set with repetitions, e.g. where each member also has a *multiplicity* associated with it. Similarly, the deletion function checks whether the value is part of the set, and if so moves the last member of the set at the value's position and shortens the size indicator accordingly.

Checking Account

A checking account is a data type that supports operations like balance inquiry, deposit and withdrawal. Our implementation will automatically keep a log of all transactions that involve the checking account.

```
;;
;; Each entry in the log file will include:
;;
(deftype account-info
  (int32 balance)
  (enum32 (CREATE DELETE DEPOSIT WITHDRAW ..)
          operation)
  (int32 amount)
  (int32 date))

;;
;; The context is a minimal one: a
;; linear map with a single entry to
;; point into the log
;;
(defcontext checking :linear (0 0)
  ((data-ref status log))
  (data log account-info))
```

```
;;
;; Operations of the type
;;
(defaccess create (amount)
  "Creates a new checking account"
  (let ((account (checking-create))
        (init (make-account-info
                :balance amount
                :operation CREATE
                :amount amount
                :date (time))))
    (checking-write account 0 init)
    account))
(defaccess balance (account)
  "Returns the current balance"
  (account-info-balance
    (checking-status
      (checking-read account 0))))
(defaccess deposit (account amount)
  "Make a deposit in the account"
  (checking-with-locked-cx account
    (seq-deposit account amount)))
(defaccess withdraw (account amount)
  "Withdraw from the account"
  (checking-with-locked-cx account
    (seq-withdraw account amount)))
;;
;; Non exported functions
;;
(defun seq-deposit (account amount)
  (let* ((last (checking-status
                 (checking-read account 0)))
         (new (make-account-info
                :balance
                  (+ amount (account-info-balance last))
                :operation DEPOSIT
                :amount amount
                :date (time))))
    (checking-write account 0 new)))
(defun seq-withdraw (account amount)
  (let* ((last (checking-status
                 (checking-read account 0))
         (new (make-account-info
                :balance
                  (- (account-info-balance last) amount)
                :operation WITHDRAW
                :amount amount
                :date (time))))
    (cond
      ((< (account-info-balance new) 0) (overdraft account new))
      (t (checking-write account 0 new)))))
```

We use here a context that does have an associated data element. We keep the current status of the account in this element. The semantics of a context-write is such that Agora adds a new value in shared

memory, and only garbage collection will remove the old, unreferenced value(s). The garbage collector can be controlled from an application like this one, so that it runs only after the logs have been retrieved and saved on disk . In this way, keeping a log of the transactions becomes a separate activity from transaction processing, with some performance gain.

3. Agora From C and C++

This chapter describes the *programmer's view* of the Agora system, as seen by the C language programmer. Not surprisingly, the basic interface is defined by a set of functions that are included in a library. This is then augmented by the functions that the compiler generates, specific to the user's defined shared data structures. Section 3.1 is an overview of the capabilities offered, and offers some hints on how to program a C agent. Section 3.2 lists all the functions available in the library. The C++ language is so closely related to C that no basic code had to be written to especially support it. To see how C++ interfaces to the Agora shared memory, e.g. via appropriate class definitions, see [6].

3.1. C Agents

It is difficult to hide anything from C: the language offers no support at all for separating the specifications from the implementations of modules [42], nor for control of visibility [16]. For a C programmer it is natural to look inside the code of the most remote function, to "see what it does". There is possibly only one exception to this, the Operating System itself (e.g. Unix, of course). In this case, a brief functional description (the specification) of the system calls that it provides seems to suffice. This approach also applies to modules that are not strictly part of the Unix operating system, e.g. the so called system libraries. In this case the interface is a set of function calls, and again they are specified via brief functional descriptions along with their C interface. It has become commonplace to provide the description of a module in the form of "man pages", e.g. more or less standardized documents that are available online to users.

This solution, e.g. extending the language with a set of well-defined and well-documented functions, is the least-costly one. Designing a new language, more or less derived from C, is definitely inappropriate in our case: we cannot call "multilanguage programming" a situation where each language has been changed to the point that it bears little resemblance to the original. An intermediate solution, e.g. a preprocessor for a limited set of well defined syntactic-semantic extensions is appealing [38], but again it is specific to each language, and possibly requires a sizeable amount of work for each language. Supporting a new language requires paying the same effort all over again.

A C Agora agent therefore is not much different from a normal sequential program. Indeed, execution is strictly sequential inside it. Explicit synchronization with other activities is provided by a single function, *wait_activation*[1]. Most functions return to the call point, possibly with errors, in a fixed amount of time. A limited number of functions can use a variable amount of time to execute, for the following reasons:

- the operation requires *O(n)* time, *n* being the size of its input. This is the case for cx_write, eval_events, asm_statistics
- the operation involves more than one machine: this can be the case for new_agent, cx_share, cx_write, cx_atomic_execute, map_update
- there is implicit synchronization: this is the case for cx_atomic_execute and potentially for shared_malloc/free.

[1]The RPC package, e.g. *a_rpc* and friends, and Set/Unset_pattern are built on top of wait_activation.

A model for a C agent is depicted in Table 3-1.

```
/* A model for structuring agents */

void parse_args( argc, argv)
int argc; char **argv;
{ command line parsing code }

void init( ... )
{ (re-)initialization code }

void cleanup( ... )
{ cleanup code }

void fi( ... )
{ code for activation tag i }

int halted = 0; /* global termination flag */

main( argc, argv )
int argc; char **argv;
{
    agora_init();
    parse_args( argc, argv);
    while (!halted) {
        switch( wait_activation( BLOCK, ANY_ACTIVATION) ){
        case 0:
            init( ... );
            break;
        case i:
            fi( ... );
            break;
        case TERM_ACTIVATION:
            cleanup( ... );
            agora_finished(0);
            break;
        default:
            fprintf( stderr, "Wrong act-type.\n");
        }
    }
    agora_finished(halted);
}
```

Table 3-1: Agent Template

This template illustrates one simple way of structuring an agent. It is applicable to any language, not to C alone. The main routine initializes the interface to Agora first, then calls a function to do the parsing of the command line switches. The global flag *halted* is available to this and to other routines to terminate the program. The value of *halted* is the value the agent returns, unless terminated normally with a termination activation (tag TERM_ACTIVATION) in which case 0 (normal termination in Unix) is returned.

After command parsing, the main loop is entered. The first activation each agent receives is the *initialization activation*, with a tag of 0. This activation is generated automatically by Agora at agent creation time. The *init* function in the example is invoked every time this activation is received. It is a wise design decision to write this function so that it always brings back the agent in the same initial state: in this way it is straightforward to restart the agent and any application it may become part of. In a

similar way, on receipt of an activation of type TERM_ACTIVATION the agent should be ready to cleanup properly and invoke *agora_finished* to exit gracefully.

The real work is done in the functions $f_1 .. f_N$, to which the main loop dispatches based on the activation tag. The agent will recognize a number of these tags, but it should be ready to deal with error situations in an appropriate way. The overall structure of this agent is therefore Dataflow-like: each activation tag triggers a specific function. Note, however, that an activation tag is not necessarily bound to *one* data structure. Indeed a function f_i could receive input from multiple places, as well as enter more complicated synchronization phases using wait_activation explicitly.

One frequent case of synchronization between two agents is the Remote Procedure Call scheme (Ada lovers would call it *rendez-vous*): the two agents exchange data among themselves in the same way as parameter passing happens in a normal procedure call. The invoking agent sends the in-parameters to the callee, who does the computing and sends back the out-parameters. In Agora, this is done using a context as the callee's request queue, and another context for each caller to wait for the replay. This case is so frequent and well-known in the literature that we wrote a simple package to support it. The longest function in the package is 9 lines long, and indeed the performance is, at it should be, better than the standard operating system message primitives. See the functions *a_rpc a_snd a_rcv* for details, and the User's Manual.

3.2. The ASM library

This section lists the functions that are available to a C programmer. For a precise specification, the reader should refer to the User's Manual. Some of the functions have been already illustrated in Chapter 5, here we have attempted to avoid repetitions wherever possible.

asm_init - Creates and initializes the Agora Shared Memory. This function is called once by the shell, creates the shared segment and initializes global state and the memory pool.

agora_init - Initializes an agent. Sets up the agent's descriptor, initializes the interface to the AgoraServer.

agora_finished - Termination. Cleans up the agent descriptor and terminates the process.

new_agent[2] - Creates a new agent. Forks a new process and possibly loads a new image, reserves a unique id for the agent, passes it the arguments list. If a hostname is provided, the agent is started on that host.

self - (macro) Finds who am I. A quick way to refer to this agent's name, unique id etc.

find_agent - Gets information about an agent. A generalization of *self*.

[2]Currently Agora creates new agents only as processes (*tasks* in Mach's terminology). Use of light-weight processes (*threads*) can, however, be supported with relative little impact on the current implementation.

terminate - Terminates an agent. Sends an activation with a distinguished tag to the agent.

shoot_agent - Brute force assassination. Needless to say, a distasteful way to terminate an agent.

set_event - Installs an event for an agent. Any time the given data structure is modified, an activation with the given tag will be posted for the agent.

unset_event - Removes one (or more) event for an agent.

eval_events - Evaluates the events for a data structure. This is called by Agora automatically as needed, but a user's Pattern Matcher may find it useful.

wait_activation - Waits till an activation is available. The agent's process will be suspended (with a sigpause(2) in Unix) until the request can be satisfied.

get_activation - Retrieves the information for this activation. Indicates which specific data structure, and which component of the data structure has changed.

agent_status - Returns the agent's status code, machine location and the number of pending activations.

regulate - Controls scheduling of the agent. On Unix, raises or lowers the process' priority.

asm_statistics - Returns a number of statistical data about the running agents.

set_statistics - Toggles statistics bookeeping. Some data is always collected, but the most expensive gatherings can be suspended.

get_shared_mem_usage - Gives the current usage of shared memory in bytes, a peak indicator of the in-use memory pool.

put_activation - Posts an activation for an agent. Includes the activation tag and an optional indication of a data structure's component.

ag_err - Prints an explanation of the latest error condition, if any.

set_agent_name - Changes the Agent's name.

cx_create - Creates a new context of the given type.

cx_share - Explicit cacheing of a remote context.

cx_destroy - Context destruction.

cx_top_map - Returns the top level map of a context.

cx_read - Reads a context's entry. An entry is a list of elements (pointers to data) and other context

addresses.

cx_write - Updates an entry in the context.

cx_write_s - Same as above, optimize for sparse data. Data elements are searched first to see if similar values are already in memory.

cx_atomic_execute - Calls a function in mutual exclusion within a given context. Can recur on the same or on a different context.

map_create - Creates a new mapping. A map is an internal component of a context used for address translation. Certain access functions may need to directly manipulate maps.

map_share - Share a map.

map_destroy - Destroy a map.

l_map_update - Update an entry in a linear map.

h_map_update - Update an entry in a hashed map.

map_deref - Dereference an address through the map.

map_underef - Un-Dereferencing, e.g. reverse mapping.

map_find - Dereferencing and optional allocation of an entry.

map_gfree - Reserves a free entry.

map_iterate - Browse through a map.

map_bounds - Bounds of the index in a linear map.

set_pattern - Sends a pattern to a Pattern Matcher. A standard interface.

unset_pattern - Asks for pattern removal.

a_rpc - Remote procedure calling (RPC) through contexts, a little package built on top of the basic primitives. Very efficient, used internally too. This function sends a request and waits for the replay.

a_snd - The sending part of an RPC.

a_rcv - The receiving part of an RPC.

a_ron - Prepare to receive-on a context. Can be considered a port-creation primitive.

a_roff - Stop receiving on a context. Port destruction primitive.

shared_malloc - Shared memory management primitives. There is no reason why a user should use them directly, but they do exists and work well.

shared_free - See above.

4. Agora From Lisp

This chapter describes the *programmer's view* of the Agora system, as seen by the Lisp language programmer. The basic interface is defined in a CommonLisp package, and makes extensive use of the *foreign-function* facility of CMU-CommonLisp [32]. Other code is generated by the compiler when translating the user's defined shared data structures and access functions. The overview in Section 4.1 illustrates how Lisp can be integrated with other components in a complete application and describes both the case in which the Lisp interpreter is used as the user's Shell, and the case in which it is used non-interactively and mixed with other agents within an application. Section 4.2 contains some practical considerations about the use of the system, as currently implemented in the CMU-CommonLisp environment on the IBM RT PC workstation.

4.1. Lisp and Alien Languages

Lisp is usually considered a closed-world by its users, and anything that lives outside Lisp is bound to live a separate and minor life. Most of this prejudice originates from the difficulties of interfacing to other languages that Lisp had in the past, but nowadays things are changing. Various Lisp implementations have introduced a more or less sophisticated *foreign-function* (note the choice of the adjective) facility that allows other code, usually C on Unix machines, to be used by Lisp [18, 12, 32].

This facility solves the problem (or at least it goes a long way towards solving it) for sequential programs, but does not help as far as parallelism is concerned. Although various research is being performed to provide primitives for parallel programming to Lisp [27, 23], this has not yet resulted in production-quality Lisps that can be put in widespread use. As far as medium and large grain parallelism is concerned, the best available facilities are packages for Remote Procedure Calling, e.g. where a Lisp process can invoke procedures that are actually executed by a different process. This is the case, for instance, of graphics applications [25] or operating system services [35].

From this point of view, Agora is an attempt to make Lisp useful for building parallel systems without changing anything of the standard Lisp semantics, and without the limits of parameter passing protocols. The Lisp program is still a sequential process, but it can interact with other processes and share very complicated data structures.

4.1.1. The Agora-Lisp Shell

One of the reasons for the good productivity of Lisp programmers is the fact that Lisp is, almost by definition, an interpretive system. This provides complete freedom for experimentation: many little functions can be easily written, tested, integrated or thrown away. This makes programming more productive because what eventually filters out to disk files is a refined set of small, flexible and highly reusable tools. The programmer has also directly available a huge amount of code that is already part of the system, and which can be constantly re-used without need for re-invention. For this reason, we not only made it possible to use Lisp from Agora, but also to use Agora from Lisp. Which is to say that one can program in Lisp both the parts of the application that must be coded in Lisp and the code that controls the overall working of the application. Since Lisp is definitely more powerful than the C-Shell, we expect a good payoff from this choice also in terms of flexibility.

The applications that will most likely benefit from the availability of a Lisp interface, besides those that are developed in Lisp or which make predominant use of Lisp agents such as in Distributed AI systems, are those where the system is composed of a user interface on a workstation and a number-crunching parallel system on a multiprocessor. In this later case, the user can benefit from a sophisticated environment on the workstation which is often not available on a multiprocessor. In addition, the load on the multiprocessor (which is still an expensive resource) is limited to the execution of the application and possibly its compilation.

4.1.2. Lisp Agents

This is an idea that a Lisper may find difficult to grasp: using Lisp **non**-interactively. Starting up a Lisp process is usually expensive but this limitation can be circumvented: if Lisp can communicate with other processes then the startup price may become acceptable, because it is distributed among various invocations. This idea is not uncommon, for instance the CMU Lisp system is being extended with the ability to connect to remote Lisp processes, and transfer to them parts of the compilation loads. In this case, the master Lisp process is able to send to slave processes (called EvalServers) arbitrary Lisp expressions that are evaluated remotely. Since the two processes do not share memory, there are limits to what can be communicated, but the flexibility is superior to simple Remote Procedure Calls.

There are cases in which it is inappropriate to choose Lisp as the primary interface for a system. For example, Lisp may not be available on a specific machine, or the implementation may be too inefficient or use too many resources on a general purpose machine. Finally, it just may be the case that other interfaces are preferred by the intended (Unix-addicted) users. Nonetheless, a growing body of software tools are developed or only available in Lisp, e.g. prototypes of AI systems, expert systems of various sorts, tools for manipulation of symbolic expressions, etc. When building an application, one would like to make use of these tools as subsystems, e.g. where it is not the final user but other subsystems that interact with the tools. Agora can solve the problem in all these cases. The tool can be transformed into an Agora agent and interfaced with other subsystems via definition of appropriate shared data structures. It will execute locally or on a remote machine as appropriate. The model for C agents depicted in Table 3-1 can easily be transliterated in Lisp, to serve as a scheme for structuring Lisp agents as well.

4.2. Notes

This section contains some notes about problems we faced in the integration of Lisp with Agora. Potential users of the system will find these notes useful since they provide a flavor of the practical aspects of the system.

All of the Agora code is contained in a single package, the "AGORA" package. Listing the exported functions and describe-ing them should provide enough documentation, we will not repeat the listing here (it can be found in Chapter 3). Care has been taken to minimize the number of exported names, so that bad interactions or conflicts with other packages are avoided. Therefore, all of the standard code that is part of the CMU CommonLisp core works as usual. All of the editor, the Common Lisp Object System, the interface to the X window manager, and all the other packages do not interfere with Agora in

any way. Memory management is also largely unaffected. Agora reserves a memory segment which is not normally used by Lisp for sharing memory with other processes. This memory segment is totally invisible to Lisp, and is garbage collected separately by the Agora Garbage Collector. The space for Alien objects, which are not garbage collected by Lisp, is kept constant by the code generated by the Agora Compiler.

There are two ways of starting up Lisp: as an Agora shell, or as an Agora agent. In the first case, there is a simple script that takes care of the various details and lets the user type simply "*agora lisp*" to get to lisp. In other words, it looks like the normal command "*lisp <args>*" is prepended with the keyword *agora*. In the second case, the user must be running the Agora C-Shell, and the command is "*new_agent lisp <args>*". Note that Lisp can obviously be started remotely, like any other agent with the "*-h <hostname>*" switch. The two cases are not identical: in the first case the script uses the Agora version of the Lisp core, in the second the default is the standard core. To get an agora agent that uses the Agora core one would use the Lisp switch "*-c*" as in "*new_agent lisp -c $ALISP/lib/alisp.core*". The idea here is that the most common use of Lisp is in acting as the primary shell, which should start quickly and have all of Agora ready. Lisp agents are most likely refined applications, which use different corefiles that contain more user code, beside Agora. Building a new corefile that contains Agora is no more complicated than usual, and a simple example script is provided.

Use of the Alien facility is not very friendly. The subtleties involved, especially in getting the best possible performance, are not easy to grasp. We are therefore implementing a more friendly interface to the user code, which works on top of the interface that uses aliens. The loss in performance appears negligible so far: the times of the basic operations are no more than doubled even without too many optimizations (about 400μsec for a write compared to about 200μsec of the alien layer, on an IBM RT-PC). The gain in usability is invaluable. Table 4-1 and 4-2 show (parts of) the code that is generated by the latest Agora compiler. The input is the specification of one of the examples of Chapter 9 (see Table 9-1).

The interface defined in Table 4-1 is simpler: two *defstruct* define all the user needs to know about the shared type, and no special operators (e.g. alien-xxx) are needed. The functions for the shared type accept an instance of a *name-service* structure and take care of all the details of transferring the data back and forth in shared memory. Note that the compiler uses static buffers for converting the Lisp structures into the Agora representation, avoiding extra memory consumption. There is unquestionably an overhead in the extra copy operation, and for very large structures (e.g. huge arrays) this may be noticeable. For this reason the Alien interface has been maintained. A user can use it directly to solve any performance problem. We have found, however, that most times the shared types themselves have pretty small representations, and that large structures are built by replicating and composing them in hierarchies. This leads us to believe that the overhead of the new interface will be acceptable in most cases.

```
;;;   File created by ATG (Agora Type Generator) version  1.30

(use-package '("LISP" "SYSTEM" "EXTENSIONS"))

;; These are the Lisp types
(defstruct handle
        (value :type 'fixnum))

(defstruct name-service
        (value :type 'handle))

;; These are the support functions

(defun name-service-write (ns address list)
"Write operation for contexts of type name-service"
  (declare (type fixnum ns)
           (type string address)
           (type name-service list))
  (alien-bind
    ((cx-tmp  name-service-write-buf c001-name-service t)
     (data-tmp handle-write-buf1 c001-handle t))
    (setf (alien-access (c001-name-service-value (alien-value cx-tmp)) 'alien)
        (alien-value data-tmp))
    (setf (alien-access (c001-handle-value (alien-value data-tmp)) 'alien)
        (handle-value (name-service-value list)))
    (if (name-service-write0 ns address (alien-sap (alien-value cx-tmp)))
      list
      nil)))

(defun name-service-read (ns address)
"Read operation for contexts of type name-service"
  (declare (type fixnum ns)
           (type string address))
  (let (c-var l-var)
    (declare (type c001-name-service c-var)
             (type name-service l-var))
    (setq c-var (name-service-read0 ns address))
    (unless c-var
      (setf l-var (make-name-service))
      (setf (name-service-value l-var) (make-handle))
      (alien-bind ((cx-tmp c-var c001-name-service t))
        (setf (name-service-value l-var)
              (alien-access (c001-handle-value
                  (alien-access (c001-name-service-value
                                 (alien-value cx-tmp)) 'alien))))))
    l-var))
```

Table 4-1: The Interface Without Aliens

```
;;;;; --- THE ALIEN WORLD --- ;;;;;

;; Alien type definitions

(def-c-record c001-handle
        (value  int))

(def-c-record c001-name-service
        (value  *c001-handle))  ;;; to a_handle

;;  Alien functions and buffers

(def-c-routine ("cx_read" name-service-read0) (*c001-name-service)
        (context        int)
        (address        asm::c-string))

(def-c-routine ("cx_write" name-service-write0) (*c001-name-service)
        (context        int)
        (address        asm::c-string)
        (list           *c001-name-service))

(defvar handle-write-buf1 (make-c001-handle))
(proclaim '(type c001-handle handle-write-buf1))

(defvar name-service-write-buf (make-c001-name-service))
(proclaim '(type c001-name-service name-service-write-buf))
```

Table 4-2: The Interface With Aliens

5. The Shell

This chapter describes the basic user interface to the Agora environment: the AgoraShell. This is a modified version of the Unix C-shell, extended to incorporate most of the Agora's functionality. Section 5.1 describes the motivations, the advantages and tradeoffs behind this choice. Section 5.2 illustrates the extra functions that are available at the shell level. Section 5.3 illustrates some examples of use of the shell, special features and limitations. For an alternative way to use the Agora environment, e.g. via the Lisp interpreter, see Section 4.2.

5.1. Extending the Unix Environment

There are many alternatives available when one wants to provide an environment for users to work and experiment with (parallel) programs. They range from the bare minimum, e.g. a single tool like a compiler for a parallel language [41] or a library of functions [40], to the overly sophisticated, e.g. language-based integrated environments [34] that allow for editing, compiling, testing, etc. all from within the same framework. On the one hand, providing minimum support leaves the beginner with many unanswered questions (although it gives a lot of freedom to experienced users). On the other hand, building integrated solutions that work well in realistic cases is still very difficult and time consuming.

We decided to pursue a solution that lies somehow in the middle: provide an extensive set of non-integrated tools, and a mean to add and combine them. In pursuing this goal, we noted that Unix is a well known and successful operating system and programming environment that follows exactly this approach. It provides for experimentation since it does not put too many constraints on the users. It also provides a very large body of building-blocks, and simple means to put them together to build more powerful tools. We decided to work inside the Unix environment, and *extend* it by modifying the **shell**, rather than hide it.

The main practical advantage we see in this approach is the ability to experiment with ideas for extending the system and at the same time limit the amount of work required and the price to be paid for failures. There are other more mundane considerations that also play a non minor role: availability of software and leveraging on other research work [4, 32, 37], portability of the final products, and a lower learning cost for users since they probably already have some knowledge about the Unix world.

The most productive programming systems, e.g. Lisp systems, allow for quick experimentation via interpretive techniques. We conjecture that it is again the building-blocks approach that makes them successful: many little functions can be written, tested, integrated or thrown away easily. This makes programming more productive because what eventually filters out to disk files is a refined set of small, flexible and highly reusable tools. The Unix C-shell brings together interpretive capabilities with other (primitive) facilities for tool integration. This may very well be the key to its success, and it definitely is one of the reasons why we chosed it.

Looking at the C-shell as an interpreter of a language makes it clear how to integrate Agora into it: integrate at the command-level the functional extensions that Agora provides to any language. The Agora shared memory then becomes a powerful mean for tool integration: not only bytes as in Unix, but

any sort of data structure can be used for communication among cooperating processes.

The C-shell provides job control capabilities via terminal multiplexing: a process can be suspended and another resumed, and at the same time the user's input is switched from the first to the second process, e.g. the *stdin* file. Files and terminal multiplexing are not precisely orthogonal: the user's terminal is a file, but other files (or pipes) cannot be multiplexed in the same way. The Agora shared memory can be viewed as an additional facility for providing I/O to a process. Any data structure in it can be "multiplexed" among agents: events (see Section 5.2.1) that are specified as *consumed* are distributed in a round-robin fashion, while normal events are replicated to all parties. Shared data structures work across the network too: there is no special case e.g. like socket or streams for standard files.

Job control over the network is still somehow a matter of research: even if Berkeley Unix has advanced networking capabilities, the core Unix mechanisms where not designed with networking in mind. Signals and terminal I/O create most problems. Rather then facing these problems, in Agora we have looked for a way to avoid them: provide a mean to start/terminate processes remotely, and then let the **X** [25] window manager take care of the interaction with the user. Starting a remote process in Agora is as simple as providing an extra switch to the *new_agent* shell command. Agent identifiers are unique and therefore all the agent primitives work transparently. This approach is different from others like **rsh** in Berkeley Unix which creates a proxy process, and **rem** in Andrew [33] which uses a combination of sockets and windowing.

Our choice of *extending Unix* obviously has the limitations that all after-the-fact solutions have since the new and old mechanisms sometimes duplicate each other or do not merge perfectly. Agora's and Unix's process control functions are not orthogonal: for instance, a process can be terminated both with a *kill* or a *terminate* command, but the semantics are not quite the same. The *new_agent* command is a redundant keyword: Agora's agent should be startable with the same syntax that is used for any other process. This is possible (e.g. putting a special *magic number* inside Agora's images), but would lead to syntactic embarrassment for starting processes remotely (e.g. to whom does the "-host <hostname>" clause belong?).

5.2. The AgoraShell

In this section we illustrate the functions that are available at the shell prompt level. All these functions are implemented as **builtin** functions of the shell, e.g. like *time, cd*, etc. They are executed interruptibly, and their arguments are **glob-ed**, e.g. allow subexpressions and template matching. The primitive for process creation (new_agent) is a special case of command execution: only the last step of calling **exec** is substituted by a call to the Agora loader.

These implementation choices lead to a smooth integration of the Agora primitives with the shell: they work as a skilled Unix user expects them to. I/O redirection, parenthesized subexpressions, template filenames, and also the CMU specific extensions of pathname search all are available within the Agora functions.

The example in Table 5-1 shows how a script could be structured in exactly the same way as a standard

C-agent is: blocking until an activation arrives, dispatching on the activation tag to various branches of the code, and terminating when a termination activation is received.

```
echo "PseudoAgent $1 started"
set running = 1
while ($running)
        set act = `wait_activation block`
        switch ($act)
        case -1:
                set running = 0
                breaksw
        case 0:
                echo "Initialization activation"
                breaksw
        default:
                echo "Activation $act"
                breaksw
        endsw
end
echo "PseudoAgent $1 terminated."
```

Table 5-1: Example Shell Script

In the following description the functions are divided into three groups: functions to start up and directly control agents, functions to browse and change shared data structures, and other miscellaneous functions that simply did not fit in any of the two above classes. The description is illustrative, and provides more an example of use than a specification. More material may be found in the Agora User's Manual.

Other functions exist that are not described here: The *CFrames* frame language is described in Section 7.1 and it is integrated in the shell as well. The *Xlisp* mini-Lisp interpreter is also available at the shell level, and is described in Section 7.2. More documentation about *CFrames* can also be found in a separate document [3].

5.2.1. Agent Control

Control in Agora is basically associated with **activations**, and that is what most functions in this section deal with. Other functions include creation of new agents and minimal monitoring tools.

Activate posts an activation for an agent. The agent will subsequently retrieve the activation and execute different actions depending on the activation tag it receives. In this sense, *activate* is like a command given to the agent, but the activator does not explicitly wait for its completion. If this is desired, the shell and the agent can synchronize at command completion, using the exchange of an activation. Other schemes can, in general, use this function as a synchronization device between the shell and other agents.

Terminate posts an activation with a special tag to the agent, with a request to terminate. This is not to be confused with terminating the agent's process (see the Unix kill(1)). On receipt of a termination activation it is intended that each agent be capable of cleaning up itself properly and then suicide. The

most common use of this function is for exiting in a clean way from an application, or in restarting specific components.

Wait_activation retrieves an activation from the shell's queue. The activation could be the first with a specific tag, or simply the one that is on the front of the queue (activation queues are FIFO queues). The operation could either be blocking or non blocking. If the request cannot be satisfied immediately the function will suspend the shell until it can be satisfied, or return a special value if the non-blocking option is specified. This function is complementary to activate, and is usually used in scripts that control an application, at synchronization points. For example, the shell could indicate that the application is about to run a new test-set, and use wait_activation to collect the indications that all the subsystems are ready.

Set_event and **Unset_event** permit control over the setting of *events* for agents. These are the preferred way to exercise control in Agora systems: an event is associated with a specific data structure. Whenever the data structure is modified the interested agent receives an activation with the specified tag, and with information about the specific change that took place. These two functions permit the shell to "configure" an application dynamically: if subsystems are connected to data structures only via events then they can be rearranged quickly, e.g. setting and unsetting of events is a mean to include or exclude components. This somehow implies that agents be written in a data-driven fashion. Other uses are related to testing of individual agents (or subsystems) in isolation, e.g. test data can be put into memory and the agent receive it in various orders.

Set_pattern and **Unset_pattern** are the standard means to communicate with a user-level pattern matching agent. A pattern is an arbitrary string that is only interpreted by the pattern matcher. The pattern matcher should activate the specified agent when the pattern is satisfied. The uses of these functions are the same as the event related ones. In this case, the pattern matcher agent is used to perform a more sophisticated management of activations, e.g. to start certain operations under conditions that can not be expressed easily with events, or to schedule or filter the activations in the most proficient way.

New_agent creates a new agent. A new process is created (possibly remotely) to execute the given image, and a standard Unix command line is passed to it. This function is clearly used to start an application or to test an agent or to invoke an Agora tool. Note that "new_agent <command_line>" is perfectly equivalent to "<command_line>" in Unix, so all uses that apply to Unix commands also apply to this function.

Lagents prints a list of all the active agents, along with information about their name, Unix identifier, Agora identifier, location status and the number of pending activations. This function is mainly used as a interactive monitoring device, but it can also be used in scripts, e.g. to provide fancy ways of identifying agents.

Statistics controls whether collection of monitoring data is done or not by an agent. Not collecting that data results in a slightly better performance of some of the core primitives. The only current use of this function is in connection with the PerformanceMonitor tool (see Section 8.2), e.g. to turn on statistics collection when the tool is started.

5.2.2. Data Manipulation

The data manipulation functions integrated in the shell are somehow rudimentary, but on the other hand the regular C-shell also supports a very limited notion of data types and variables.

Ltypes gives a list of the type descriptor that are currently loaded in the Agora's shared memory. This is usually a check done at startup time, when each subsystem uses this function to make sure that the types it uses have been loaded, and load them if necessary. It can also be used as a debugging aid.

Create is used to create a new instance of a data structure. The main use of this function is to set up the data structures before starting the actual agents, and allocate them on the various machines. This makes subsystems more easily restartable, since they can assume that the data structures they need have been set up already.

Share caches an existing data structure on the shell's machine. This may be necessary as a simple debugging aid, or to speed up subsequent operations.

Lcolls prints a list of all the existing data structures, along with their type. It is mainly a debugging aid, but can be used in smart configuration scripts.

C_count returns the number of elements in a data structure. Can be used as a trivial synchronization device, e.g. associating each element with the completion of some operation.

Gc controls garbage collection, see Section 8.1.

Read_element prints the content of a data structure. The output is not pretty printed (as it probably should), but this suffices for simple data exchanges and debugging: not much computing is done by the shell.

Write_element adds an element to a data structure. This is often used to pass command lines or parameters to an already running agent, without restarting it. Useful both in scripts and interactively.

5.2.3. Miscellaneous

The following functions give some control over the shared memory, both on the local machine and over the network (see also Chapter 6).

Login_parameters configures the local instance of the AgoraServer (see 6.2) and if necessary starts it. It provides the server with information about the account and password to use for starting processes remotely, as well as other optional information about the location of the Agora tools subtree and dynamic memory configuration (see 5.3). This request can be re-issued, e.g. to use different accounts on different machines.

New_machine connects a new machine to the shared memory. A new instance of the AgoraServer (see 6.2) is started on the machine, if not already started. This function is used to better allocate in the startup phase the overhead of connecting new machines. Its use is optional, since the new_agent function calls it

if necessary, but recommended.

Buffer_tune controls certain optimization parameters of the AgoraServer.

Sh_mem_usage returns the amount of memory that the system is using, both for system data and for the shared pool. This is a peak indicator, to be used in conjunction with dynamic memory configuration.

5.3. Living in Agora

The AgoraShell can be, and indeed is used as a login shell for everyday computer life. To do so, at CMU a user sets the file /usr/misc/.agora/bin/agora as his login program, or alternatively executes it from his .login script. This is a shell script that sets the environment variable AGORA to refer to the root of the Agora tool tree (at CMU this is /usr/misc/.agora), adds it to the list of search paths, and then loads in memory the AgoraShell image and passes control to it. For the die-hard Lisp programmers, the script can be used to startup the Agora Lisp core instead (see 4.2 for more details).

The same procedure applies to starting the Agora shell from a normal Unix shell. The Agora shell can also be set as the default interpreter for shell scripts, in this case however each new shell instance will create a separate shared memory space for itself and its subprocesses.

The Agora shell is slightly more demanding on the memory subsystem than the normal C-shell: the image is about 1.5 times larger, and it preallocates the whole shared memory segment at startup time. The size of the shared segment can be defined at startup time, and the default of 8 megabytes is largely pessimistics, e.g. it is plenty.[3] We have found that our largest application [1] seldom uses more than 3 megabytes of shared data. The shared segment can also be allocated anywhere in the processes' address space, and again this can be specified at startup time. The only limitation to the user's fantasy is given by addressability limits that an architecture may have, e.g. Vax architectures provide full 32 bit virtual addresses while the Encore Multimax provides 24 bit virtual addresses. It may also not be a wise idea to overlap the shared segment with e.g. the stack segment!

---

[3] The choice of pre-allocation of memory is largely due to the rules for sharing of memory in Mach, e.g. via inheritance from the parent only. We do not feel these rules are unduly restrictive.

6. The Networked Shared Memory

This chapter describes how the Agora shared memory is implemented on a set of machines connected by a Local Area Network. The network extension is completely transparent to the user, and is realized via separate agents, the **AgoraServers**. Section 6.1 describes the key ideas on which the extension is based. Section 6.2 explains in detail how the server works and how it takes advantage of the features of the Mach operating system for its operation: the ideas described here apply to other environments as well. Section 6.3 contains other notes on the various issues that we had to face during the implementation.

6.1. Memory and Messages

Agora uses the shared memory abstraction of the Mach operating system [4] to share data structures among agents that are on the same machine. This approach works on both uniprocessors and shared memory multiprocessors. The only operating system support that is needed, besides shared memory, are simple locks (on Mach these are implemented in user space by using test-and-set machine instructions) and sleep/wakeup (implemented by using Unix signals).

Things are more complicated when dealing with loosely-coupled architectures. Work in distributed systems has concentrated mainly on message-based models, and few have attempted to use memory-based models [24, 14, 30]. The main problem is that it is hard to efficiently maintain coherence when the communication bandwidth is low. Agora's implementation demonstrates that this is not only feasible, but even more efficient than the message-based approach in various cases of practical importance.

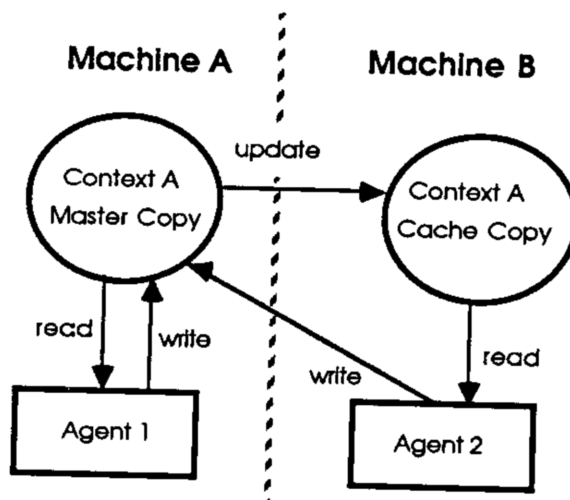

Figure 6-1: Sharing Contexts Network-wise

In Agora, a data structure (context) is stored in the shared memory of the machine of the agent that created it. It is then replicated in all the machines containing agents that use it, in a copy-on-reference fashion. When a context (*master copy* in Figure 6-1) is shared by an agent on a different machine, a complete copy is made on that machine (*cache copy*). Read operations refer to the cache copy, while write operations are forwarded to the master copy. The AgoraServer on the master copy's machine is

responsible for notifying other AgoraServers when a local context with remote cache copies is changed. The AgoraServer on the cache copy's machine sends write requests to the appropriate server. See [6] for a discussion of the performance of this and other approaches.

This approach obtains optimal performance (lower bound on the number of messages) in many practical cases. For instance, in the case of *(N-1) producers, 1 consumer*, the number of messages required is only (N-1) since the master copy can be associated with the consumer if the producers do not need to re-read the data they produce. Moreover, in Agora, data structures can be allocated on different machines without loosing global visibility. This lets the user optimize performance by carefully partitioning and allocating the *master copies* in a balanced way.

6.2. The AgoraServer

Many of the functions provided in the ASM library may need to operate on entities that live on some other machine, like agents or contexts. Remote operations are therefore implicit in the definition of the primitives themselves. A well known approach to remote operations is to use a server that executes the operation remotely on behalf of the caller. In our case this is the task of the AgoraServer. Each ASM function has been coded according to the scheme: *if the object is local then perform the operation, else if the AgoraServer is running then ask it to perform the operation remotely, if not then return an error*.

Each ASM function call can therefore become a remote procedure call to an AgoraServer. In that case, the Agora facilities for remote procedure calling are used to notify the local server of the request. The server then transmits the request to the fellow server on the appropriate machine. The remote server executes the operation, and sends the reply back to the local server. This in turn sends the reply to the agent. This process is illustrated in Figure 6-2. Most operations do not require a reply, and in this case we take an optimistic attitude and do not use acknowledgements. The AgoraServers communicate among them by using Mach's network communication primitives. Any transport mechanism that provides similar reliable transport of data between processes on different machines (e.g. [39] [13]) is also perfectly appropriate.

There are two motivations for this scheme: economy and portability. Economy calls for reusing ideas and code wherever possible: why use a different RPC facility in the library to hop over the net when one is already available? Portability requires that details about the inter-machine communication primitives offered by the operating system be hidden and confined to a single place, and not spread throughout the code of the ASM. Should we decide to use a different protocol we only need to recode (and re-debug!) parts of the server. The AgoraServer itself is implemented like any other agent although its algorithm is based on information about the internal structure of the shared memory that is normally hidden from user agents. The price we pay by using a single server per machine is serialization, which partially limits the available parallelism[4]. On the other hand, we have taken advantage of this serialization by buffering network communications to get a better performance, e.g. packing multiple requests and data towards

---

[4]This should not be overestimated: the network interface is a bottleneck anyway. In Mach there is also another serialization point (and extra context switching on uniprocessors) in the Netmsgserver, e.g. the single process that extends the IPC capabilities over the net.

the same machine in a single message.

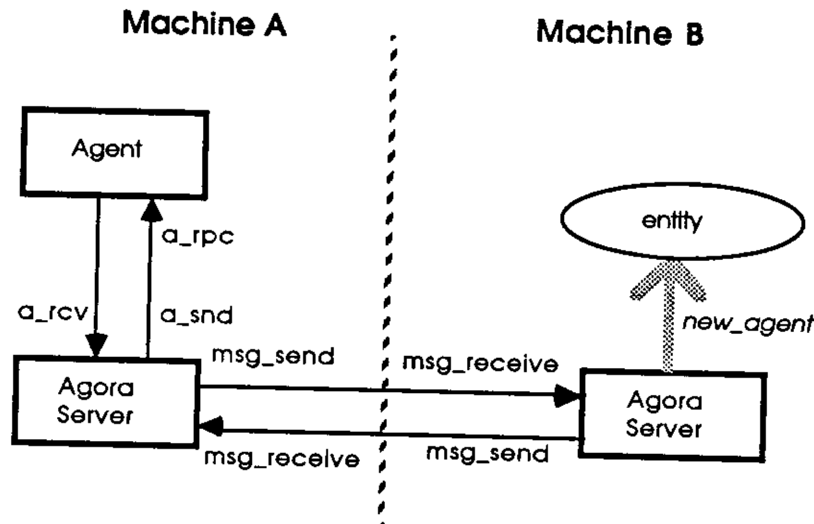

Figure 6-2: Example of Remote Operations

Mach Dependencies

The server takes advantage of some features of the Mach operating system for its operation. These are the compatibility with Unix, and the interprocess communication (IPC) facilities. The server was designed with multi-threaded control, but the actual implementation used Unix's interrupt handling rather than Mach *threads* since at the time threads were not available.

The server has a main control thread that, like any other agent, waits for activations. An activation can only be a remote procedure call from some other local agent, and it is dealt with accordingly. A second thread is activated when messages are sent to the server from other fellow servers: these are requests from remote agents, and again they are dealt with as appropriate. A third thread is activated at fixed time intervals (default every 5 seconds, but dynamically settable), and checks the status of the buffers and flushes them when appropriate.

When the server starts, it registers an IPC *port* with the Mach network name server. The name is composed of the user's id, shell's process id, machine name and a standard prefix. In this way it can be reconstructed by any other server, and it does not prevent a user from running multiple separate Agora applications at the same time. That port is then used by other fellow servers to send requests to. The IPC facilities are otherwise used as a *transport* device: there are only two types of messages that are exchanged: remote procedure call messages, and data transfer messages. The first type is a two-part message, with a small header of integer parameters, followed by string parameters. The second only contains raw bytes, and the Agora type descriptors are used to interpret it.

When using remote procedure call schemes, attention must be paid to *nested* remote procedure calling: it is easy to fall in the *deadlock* trap, e.g. when a process generates a request to a second process while the latter is still waiting for a reply from the former. In our case, we have found unavoidable to use nested remote procedure calls because AgoraServers are peers among them. To solve the deadlock problem the

server takes advantage of a feature of Mach's IPC that allows a process to wait for a message from more than one port at a time. In this case, both the reply port and the normal request port are listened to. When the server needs a reply, it enters a short loop that receives on both ports. If the message arrives on the request port it is dispatched to the service function, otherwise the loop is exited since the correct reply has been received. One noticeable side effect of nested remote procedure calls is a strong push toward a functional programming style, e.g. avoiding global state and side effects.

6.3. Notes

Naming

An important problem on loosely coupled multiprocessors is naming, e.g. how information can be retrieved starting from an object's identifier. Agora uses a flat naming space to suit its shared memory based model. Each agent or data structure has a unique identifier associated with it: all is needed to operate on an object is to know its identifier. If the associated object cannot be found on the local machine, it is considered remote and the request is handed over to the AgoraServer. There are three ways for finding on which machine an object lives: include the location of the object in the object identifier, send a message to one particular server which has global information, or interrogate all the other servers.

If the identifier of an object includes the identity of the processor that created it, then all operations can be sent directly to that processor. This approach works well for ensemble machines e.g. [36], and all those cases where there is a precise notion of a (short) processor identifier. If name resolution is infrequent (as it is the case in Agora, since the location of an object does not change), the second approach becomes appealing. Agora uses this second scheme. The server which is responsible for name resolution resides on the user's machine. This server already has most of the necessary information available since it is the first one that is started and the one that starts (most of) the others. This server is also in charge of creating global identifiers and distributing them on request. Performance becomes an issue only when the number of machines grows *and* the application is very dynamic. In most cases creation of new objects is easily relegated to the application startup phase. Once the location of an object becomes known, each AgoraServer notes it in a lookup table and subsequent operations go directly to the destination server. The third scheme causes heavy message traffic if broadcast messages are not available. Even if broadcast were available, this scheme would cause all the servers to waste processing time to process each request.

Data Translation

Moving binary data between different machines has always been a difficult operation, escaping any attempt to set standards. The AgoraServer provides an efficient algorithm to translate typed data across heterogeneous machines at runtime. Common algorithms either require a type description that is compiled with the code (because it is a procedural description) [39], or require type information that is transferred with the data in each message [35]. In Agora, instead, each machine marks messages with a machine tag, and messages between identical or compatible machines do not require translation. When translation is necessary, *the destination machine* uses the runtime type descriptor (see Chapter 2) associated

with the destination context to convert the data into the appropriate format.

Besides conventions on the packing of data into messages (e.g. the first word is the identifier of the destination context, etc.), the conversion procedure uses an algorithm which is one-pass and table driven. Data is converted in-place to avoid wasting memory. The core of the algorithm is summarized in Table 6-1.

```
/*
 * Translates a 'buffer' of 'tot' elements for context 'cx'
 */
int translate(cx, tot, buffer)
CONTEXT cx;
int tot;
int *buffer;
{
    ....
    descriptor = find_type( cx );
    while (tot--) {
        num_bytes = descriptor->tot_size;
        if (num_bytes == 0)      /* variable sized data */
            num_bytes = swap_32( *buffer );
        i = 0;
        while ( (num_bytes > 0) &&
                (i < descriptor->num_fields) )
            if (do_one(descriptor, &i, &buffer, &num_bytes))
                return(-1);
    }
}

/*
 * Translate one element.  Possibly recurs for nested records.
 */
int do_one(descriptor, p_index, p_buffer, p_counter)
type_descriptor *descriptor;
int *p_index, *p_counter, **p_buffer;
{
    ...
    switch( descriptor->field[*p_index].field_type){
    case type_tag:
        t = descriptor->field[*p_index].num_elements;
        for (j = 0; j < t; j++)
            swap bytes as appropriate
        break;
    case BEGIN_RECORD:
        f = s = descriptor->field[*p_index].num_elements;
        t = descriptor->field[*p_index].num_fields;
        do {
            for (j = 0; j < t; j++)
                if (do_one( descriptor, p_index,
                            p_buffer, p_counter))
                    return -1;
        } while (f ? --s : (*p_counter > 0));
        break;
    default:
        return -1;
    }
    (*p_index)++;
    return 0;
}
```

Table 6-1: Data Translation Algorithm

The type descriptor contains various information, but the routine *translate* uses only the number_of_fields and the total_size. Note that variable sized elements are marked with a 0 in the total_size field. Each field is translated by the function *do_one*. This function dispatches on the field type to different sections of code, one per basic type. For each basic type, the code includes a loop to account for arrays of the basic type, and inside the loop instructions that

1. align the buffer pointer as appropriate to the type e.g. 32 bit entities are always transmitted aligned on 32 bit word boundaries;
2. swap the bytes as appropriate;
3. update the pointers and counters as appropriate.

The only special case is for floating point numbers. Two representations are currently recognized: Vax and IEEE standards. Both have a short 32 bit format and a longer 64 bit one. The short formats are largely compatible, the longer ones will cause truncation in the IEEE –> Vax direction given the smaller range for the exponent on the Vax, and a very little loss of precision in the Vax –> IEEE direction because of the smaller mantissa.

A different case is the case of a record field. Here the function recurs for each sub-field: notice that the type descriptor is in-line expanded by the compiler, with bracket field descriptors for begin and end of a record. The code may be a little obscured by the side-effects used in this recursion, but the basic idea should be clear.

Security

To operate correctly, a server like the AgoraServer must be capable of starting up itself on remote machines, and of providing appropriate accounting information while doing so. Lacking *process migration* capabilities, it is important that owners of the various machines be able to exercise some control over the utilization of their resources. Under Mach, Agora uses a system provided server, the **ipcexecd** server, to perform this operation: a request is sent by an AgoraServer to the ipcexecd server to start a copy of itself on the machine, under a certain account[5]. The accounting information is verified via the Unix **su** program, and if the request is granted the new server is started. The new and the old AgoraServer then establish a communication link among them with the help of the Mach-IPC MessageServer. This scheme is of general applicability: whatever the communication device, a copy of the AgoraServer could be started at boot time on each machine, and it would act in exactly the same way as the ipcexecd server acts under Mach. The communication link among AgoraServers is both private (the ipcexecd server is only involved in the startup phase, not in the actual communication) and protected. These properties are obviously desirable in any other IPC facility as well.

Bottlenecks

The bandwidth that the Ethernet offers to the final applications under Unix is not impressive. Therefore special attention has been paid on how the AgoraServer uses the transport medium, to get the

[5] The accounting information is explicitly given to the AgoraServer by the user, as encrypted information in a remote procedure call (see **login_parameters** in 5.2). It is maintained and transmitted in encrypted form, and not stored on disk files.

best available performance. The resulting algorithm takes into account both time and space tradeoffs, and moreover it can be tuned dynamically by the user. The server maintains a buffer of outgoing requests for each destination machine. A buffer is flushed and an actual message is sent if

1. time is at least **ipc_tick** milliseconds after any previous send operation and
2. time is at least **2 * ipc_tick** milliseconds past the last send operation to the same machine or
3. its size has become larger than **ipc_size**

The goal is to avoid clobbering the local network interface, leave enough time to other servers for processing *and* making requests, and pack as much information as possible in every single message, without too much latency. The values of **ipc_tick** and **ipc_space** can be dynamically adapted to the expected performance of the IPC transport, which is an ever changing entity with the current networking software.

7. Agora-Extras

This chapter describes some non-core components that are available in the Agora system in form of linkable libraries and Lisp packages. They include *CFrames*, a small distributed frame language derived from SRL [22] and *Xlisp* [5], a small Lisp interpreter that is linkable within a C program and allows for evaluation of CommonLisp expressions. Section 7.1 describes in detail *Cframes* and Section 7.2 describes *Xlisp*.

7.1. CFrames

CFrames[6] is a miniature frame system initially written in C. It implements the basic frame language functions for creating a semantic network with frames, slots and values, and for searching the network using multiple inheritance. The system is streamlined in that it only uses "is-a" relationships for inheriting information from other places in the network. It automatically maintains inverse relations, and supports multiple inheritance along the "is-a" relation. When a frame has multiple parents, inheritance searches are performed in a breadth-first manner over the list of parents.

CFrames was developed as a medium through which a collection of intelligent tools could interact while developing large-scale distributed systems. A substantial portion of the knowledge necessary for constructing and executing these systems can be stored in a semantic network and manipulated by the *CFrames* tools. Since *CFrames* has been implemented on the Agora shared memory, the network can be shared by a number of tools over a number of machines, thereby facilitating distributed system development.

CFrames has been incorporated into the AgoraShell, as well as in a linkable C library, and subsequently in a Lisp package. It has been compiled under the Mach Operating System, and must be used within the Agora environment. Once inside the Agora shell, using *CFrames* is as simple as typing single command lines. Using *CFrames* in a C program requires including one .h file and linking in a single library. In Lisp, *CFrames* is a separate package, and is part of the Agora Lisp corefile.

CFrames consists of a number of functions, which can be accessed either through the shell or from a C or Lisp program. To simplify the use of the language, a naming scheme for the functions has been devised, one which relies on one-letter mnemonic prefixes to stand for basic operations vis-a-vis the semantic network. They are:

Prefix	Meaning
a	append
c	create
d	delete

Prefix	Meaning
g	get
p	print

Prefix	Meaning
r	read
w	write

Currently the only data types *CFrames* understands are structures called "a_node," strings, and integers. This restriction is not seen as terribly constraining, although other types may be added in the

[6] Design and implementation of *CFrames* was done by R. Lerner and M. Bauer. The user interface of *CFrames* was influenced by the KR system of D. Giuse.

future. A_Node encapsulates all the relevant information needed for a node in the network. Its internal representation is unimportant, simply note that a number of C functions return references to such structures and require them as arguments. When talking to *CFrames* from the shell, string and integer types are signaled by immediately prefixing the value with an S or an I. In a program, the types STRING and INTEGER have been pre-defined in cf.h. As a fine point, case is maintained for all strings injected into the network, but it is ignored for purposes of comparisons.

The next section describes a sample of the functions in detail. The goal here is only to provide a concrete understanding of some relevant *CFrames* functions. The exaustive description can be found in a separate document [3]. Each description is split into three parts. The first is a brief description of what the function does. Then the Syntax shows how to invoke the function from a C program or from the shell. The Lisp syntax can easily be inferred from the shell one. Finally a sketchy example of use of the function is given.

7.1.1. Language Overview

Cframe : Create a frame.

The function *cframe* creates a frame with the given name. If a frame already exists with that name, nothing happens. In C, The value returned is a pointer to the actual frame structure if created or found, and NULL otherwise.

C Syntax:

```
a_node *
cframe (framename)
char *framename;
```

Shell Syntax:

```
cframe framename
```

C Example:

```
a_node *pframe;
pframe = cframe ("Parser");
```

Shell Example:

```
cframe Parser
```

CSlot : Create a slot in a frame.

The function *cslot* adds a slot to the specified frame with the given name. If the slot already exists nothing happens. The value returned is a pointer to the actual slot structure if created or found, or NULL if the frame is not found. For convenience, if the slot does not exist in the given frame it is created. *cslot* always creates the slot in the current frame. It does not look to see if the slot can be inherited first.

C Syntax:

```
a_node *
cslot (framename, slotname)
char *framename;
char *slotname;
```

Shell Syntax:

```
cslot framename slotname
```

C Example:

```
a_node *pslot;
pslot = cslot("Parser", "IS-A");
```

Shell Example:

```
cslot Parser IS-A
```

Cvalue : Replace the value(s in a slot with a new value.)

The function *cvalue* replaces the value(s) in a slot with a single new value. The value is specified by a type, and a pointer to the data. The type describes the data. There are currently only two types of data, namely STRING and INTEGER. In C these are integer constants defined in cf.h. These must be used in any C calls requiring type specification. In the shell, the types are specified by immediately prefixing the value with either an S or an I. The pointer in C is always a character pointer pointing to the data. Upon execution of this function the data is moved into a shared data structure. The number of bytes copied depends on the type. An entire string is always copied, sizeof(int) is always copied for an integer.

In C, the value returned by *cvalue* is a pointer to the actual value structure, if created and NULL if the frame does not exist. For convenience, the slot is created if non-existent, as in *cslot*. *cvalue* always places the value in a slot in the current frame. It does not try to find the slot through inheritance.

C Syntax:

```
a_node *
cvalue (framename, slotname, type, pdata)
char *framename;
char *slotname;
int type;
char *pdata;
```

Shell Syntax:

```
cvalue framename slotname SIvalue
```

C Example:

```
a_node *pvalue;
pvalue = cvalue("PARSER", "IS-A", STRING, "agent");
```

Shell Example:

```
cvalue parser is-a Sagent
```

G1stvalue : Gets the first value in a slot.

The function *g1stvalue* finds the specified slot through inheritance and returns a pointer to the first value node in the slot. The actual data represented by the value node is returned through the pointer parameters. If the slot or frame is not found, or there were no values, the function returns NULL and the pointer variables are all set to NULL. The inheritance search is performed as follows:

1. If the slot is in the current frame, return the first value.
2. If frame has an IS-A relation, each frame specifed in the is-a relation slot is recursively searched in turn.

The actual value is returned by three variables, the type,the size and a pointer to the data. When calling

g1stvalue the address of a character pointer should be given for these parameters. The address of an integer should be given for the size and type parameters. To get to the next value in a slot, the value structure which *g1stvalue* returns must be handed to *gnxtvalue*.

Note that the shell version of the retrieval functions are a bit different. The value is simply printed out, and a reference is maintained internally to the value node returned.

Shell Syntax:

g1stvalue **framename slotname**

C Syntax:

```
a_node *
g1stvalue (framename, slotname, type, size, pdata)
char *framename;
char *slotname;
int  *type, *size;
char **pdata;
```

C Example

```
a_node *pvalue;
char   *pdata;
int    type, size;
pvalue = g1stvalue("Parser","Directory",&type,&size,&pdata);
```

Shell Example

```
g1stvalue Parser Directory
```

Gnxtvalue : Gets the next value in a slot.

The function *gnxtvalue* finds the next value in a slot. It takes a pointer to a value node in the slot and returns the next value in the list. It returns things in exactly the same manner as g1stvalue.

Shell Syntax:

gnxtvalue

C Syntax:

```
a_node *
gnxtvalue (pvalue, type, size, pdata)
a_node *pvalue;
int  *type, *size;
char **pdata;
```

C Example

Shell Example

```
a_node *value;                          gnxtvalue
char *type, *data;
int size;

for (value = glstvalue("Parser","Sources",&type,&size,&data);
     value != NULL;
     value = gnxtvalue (value, &type, &size, &data))
{
  if (type == STRING)  printf ("%s\n", data);
}
```

String_ValueP : Finds a string value in a slot.

The function *string_valuep* determines if the given string value is a member of the specified slot. It returns TRUE if it found the string by inheritance and FALSE otherwise. Only values whose type is STRING are looked at, and the search is not case sensitive.

Shell Syntax:

stringvaluep **frame slot string**

C Syntax:

```
string_valuep (frame, slot, string)
char *frame;
char *slot;
char *string;
```

C Example

```
if (string_valuep ("Parser", "Sources", "verifier.slisp"))
  printf ("The verifier is included in the Parser.\n");
```

Shell Example

```
stringvaluep parser sources verifier.slisp
```

RTest : Determines if some relation holds.

The function *r_test*, starting at `frame`, searches the network through the `relation` slots, trying to find the `goalframe`. If it locates the `goalframe` it returns TRUE, otherwise, it returns FALSE. The search starts at `frame`. If the current frame is the `goalframe`, TRUE is returned. Otherwise, each of the frames whose names are values of the `relation` slot, are checked. This is a depth-first search. Note that inheritance is not used to find the `relation` slot in a frame. The `goalframe` does not actually need to name a frame. The search succeeds whenever the string value `goalframe` is a value of the specified `relation` in one of the frames being searched.

Shell Syntax:

rtest **frame relation goalframe**

C Syntax:

```
r_test(frame, relation_slot, goalframe)
char *frame;
char *relation;
char *goalframe;
```

Shell Example

```
rtest parser is-a agent
```

C Example

```
if (r_test ("Parser", "IS-A", "Agent"))
  printf ("The Parser is an agent.\n");
```

7.2. Xlisp

As its author defines it [5], "*XLISP is an experimental programming language combining some of the features of LISP with an object-oriented extension capability*". It is completely written in C, and therefore easily portable on a large class of machines. The syntax accepted is, to a large extent, a subset of CommonLisp. Needless to say, it is the CommonLisp compatibility that made it interesting to us, not the object-orientedness, which is limited and buggy, at least in the version we have. Other limits relate to efficiency: the interpreter is slow and there is no compiler to alleviate it. In the Gabriel benchmarks, XLISP is roughly 100 times slower than the CMU-CommonLisp interpreter.

Nonetheless, we found it perfectly suited to our needs. Integrating a small Lisp interpreter in C programs solved the problem of handling a database of procedural knowledge, e.g. what was missing in *CFrames* to give us a distributed object database [7].

It was easy to modify the original code of the interpreter and turn it into a linkable library, which exports a single function that takes a string and evaluates it as a Lisp expression. In this way, any agent can look at the database, retrive a method associated with an object, and execute it interpretively. The added power was enough to turn a simple graphic interface to the database into an small object system! Slightly less successful was the attempt to convert the C-shell into a Lisp-shell, integrating Xlisp into it. The major gain in this case was the ability to develop interpretively methods on any machine... we never really dreamed of converting programmers from C to Lisp! Nonetheless, a script for the AgoraShell can freely mix C-like expressions with Lisp-like expressions, for example when the true scoping of variables and better data types of Xlisp make it more adapt to express the algorithm.

Xlisp is interfaced to the shell, and likewise to any program, via four functions:

- **xleval** takes a single string parameter and interprets it as a Lisp expression
- **read_eval_print** enters an interactive loop of reading from standard input a Lisp expression, evaluate it, and print the result to standard output.
- **init_fountain** loads in the basic set of default methods.
- **invoke** uses method invocation to apply a certain operation to an object, e.g. the method is looked up through inheritance, and parameters are likewise avaluated.

Most of the usual Agora functions are internally available to Xlisp. In fact, any function that has been linked in the executable image can be invoked via the *call* function, provided the image has not been

stripped of the symbol table.

8. Other Tools

This chapter describes various tools that are part of the Agora programming environment. Although they are described together, they all tackle separate issues. Section 8.1 describes the GargabeCollector, a user-level agent that incrementally and in parallel recovers reusable space from the shared memory. Section 8.2 describes the PerformanceMonitor, which monitors the status of the shared segment on the various machines and graphically depicts the evolution of the key performance parameters. Section 8.3 illustrates how complex pattern matching can be implemented at the user level by a specialized agent, the PatternMatcher. Section 8.4 is devoted to debugging. Section 8.5 describes a utility to save-restore in disk files the content of the shared data structures, in a machine independent format. Finally, Section 8.6 describes the user-level loader that Agora uses to load agent images in memory.

8.1. Garbage Collection

For performance reasons, data memory is largely managed in a write-once fashion in Agora. This makes it necessary to recover reusable storage, and to this end Agora provides a **garbage collector** agent. This agent can can be started by a user at any time, and it works incrementally and in parallel without interference with the running application.

Whenever data is moved from an agent's local memory to the shared segment, a new piece of memory is allocated from a shared pool. This happens in the primitive **cx_write**. The new object can then be referred to by more than one map, using the **map_update** primitive. The same primitive will also remove references to shared objects, and when the last reference to an object is removed, its storage space can be recovered.

There are two efficient schemes for garbage collection that are applicable in our case: reference counting, and mark-and-sweep. Both can be applied, and both have advantages and disadvantages. With reference counting, each call to map_update decrements and checks the counter associated to the old pointed-to object. If an object becomes unreachable, it is collected. The collection cost can be minimized using a separate agent that puts back the objects in the shared pool, possibly attempting compaction. With mark-and-sweep, a first pass is done through all the maps to mark the objects that are still accessible, and in a second pass all unmarked objects are reclaimed. The algorithm can be executed by a single separate agent, and parallel variations are also well known.

The first algorithm minimizes the amount of memory in use at any given time, since unaccessible storage is identified immediately. The price is a fixed overhead in the map_update primitive, possibly larger if each agent gives back memory immediately, rather then relaying on a separate collector agent. There are also other costs, more difficult to measure, associated with interference between agents in e.g. decreasing the reference counters (a non atomic operation on most processors).

The second algorithm can easily be implemented with a separate agent, and requires no overhead on the standard primitives. The agent could run continuously without minimizing memory usage. On the other hand, it will not cause interference with other agents since there are no synchronization constraints on e.g. reference counters. Moreover, a single bit per object constitutes all the required space overhead. The garbage collector agent will, however, use much more cpu cycles than in the previous case.

We decided to use a mark-and-sweep scheme for the Agora GarbageCollector, mainly because it does not cause extra interference among the components of the application, and because it is more controllable. The user can in fact decide how frequently the algorithm is run, prevent it from running at inappropriate times, or even run it in at appropriate times e.g. within successive executions of the application. When an application runs on multiple machines, a separate instance of the GarbageCollector agent runs on each machine. The various instances are totally independent of each other. A memory object could become unaccessible on one machine, and its copy be still accessible on another machine: this depends on which specific contexts have been cached on the specific machine.

8.2. Performance Monitoring

Sometimes it is easy to understand what a parallel system really does during the execution, sometimes it is not. What is important to understand in all cases is where the performance bottlenecks are, even when the system performs satisfactorily. Agora provides both simple monitoring devices, e.g. shell level commands like *lagents* and others described in Chapter 5, and a **Performance Monitoring System** described in this Section. The purpose of the tool is to display on a screen, both numerically and pictorially, the evolution of the key performance parameters of an Agora/Mach application. Since the tool uses **X** [25], the display can be located on a remote machine. These statistics are collected by the Performance Monitor itself, without modification of the running system, and with very little overhead.

What Is Displayed

Figure 8-1 shows what the performance monitoring window looks like on a black and white screen. The display is organized as a matrix, each row is dedicated to an agent and shows:

NAME: the agent's name;

AGORA ID: its (unique) Agora identifier;

UNIX PROCESS ID: its (non unique) Unix identifier;

STATUS: an agent can be in one of the following conditions:

- *waiting*: waiting for an activation;
- *running*: running ok;
- *terminated*: terminated without problems;
- *terminating*: received a request to terminate, but still running;
- *shot*: the agent's process has been (brutally) killed;
- *limbo*: not started yet;

LOCATION: the name of the machine on which it is running;

USER TIME: the total amount of time spent executing in user mode;

SYSTEM TIME: the total amount of time spent by the operating system executing on behalf of the agent;

UNIX SIGNALS: the number of Unix signals received;

AGORA SIGNALS: the number of signals delivered by Agora;

VOLUNTARY CONTEXT SWITCHES: the number of times a context switch resulted to the agent voluntarily giving up the processor, before its time slice was completed (usually to await

availability of a resource);

INVOLUNTARY CONTEXT SWITCHES: the number of times a context switch resulted due to some higher priority process becoming runnable or because the agent exceeded its time slice;

ACTIVATIONS: the number of activations received;

PENDING ACTIVATIONS: the number of activations still available in the queue and not used yet;

MEMORY USED: the total amount of Shared Memory in use (peak value);

WRITES DONE: the number of times the agent wrote into shared memory;

BYTES WRITTEN: the total amount of data moved into shared memory;

READS DONE: the number of times the agent read from shared memory;

BYTES READ: the total amount of bytes accessed/ible in shared memory.

00:00:000 **Performance Display**

Name	UID	PID	Status	Location	UTime	STime	Sig	AgSig	VCSw	IvCSw	Act	PAc	Memory	Writes	WBytes	Reads	RBytes
pm++	2	8636	Waiting	SPEECH2	590	1660	248	57	281	258	35	0	441440	137	127476	167	154883
perf	3	8638	Waiting	SPEECH2	240	1000	77	70	87	163	1	0	211572	76	101308	7	364
fperf	4	8637	Running	SPEECH2	13450	5630	121	70	915	703	185	20	509308	10	1584	194	200448
mouse	5	8642	Running	SPEECH2	80	360	0	1	15	25	1	0	185216	32	768	0	0
perf	6	330	Waiting	LAB	100	300	68	61	78	138	1	0	203380	126	69632	6	296
scan	7	331	Waiting	LAB	80	380	126	45	140	86	1	0	257488	11	992	93	545
perf	8	8346	Waiting	SLOOP1	350	690	52	44	65	112	1	0	203380	92	63256	6	296
teal	9	8347	Waiting	SLOOP1	40	530	43	40	52	33	2	0	171612	2	400	2	24
spinit	10	8670	Waiting	SPEECH2	60	800	27	19	38	55	1	0	211744	7	672	7	84
spstart	11	8675	Running	SPEECH2	510	1490	44	12	62	198	2	0	1303100	3	12	0	0
disp	12	8676	Terminated	SPEECH2	70	690	3	1	19	40	2	126	1307216	7	1504	7	252
spag	13	8678	Waiting	SPEECH2	70	670	22	18	32	69	1	0	204776	1	304	1	12

Figure 8-1: Example of a Performance Display

All numbers that change with time (e.g. the user time) are displayed together with percentile bars that highlight which agents are using lots of resources and which are not. These numbers can be expressed either as cumulative values, from the time when the agents started, or as peak values, from the last sampling time. The type of values displayed (cumulative or peak) is indicated by the type of bar used: thick bars indicate cumulative values and thin bars peak values. The bars become different background shades of color on color displays, and the difference between cumulative and peak values is indicated by different scales of colors.

Figure 8-1 shows what can be seen on a black-and-white monitor when all cumulative values are

displayed. The application system includes agents running on three different (and heterogeneous) machines.

Other features of the Performance Monitor include:

- a history mechanism, to review what happened during the execution of a system;
- hardcopies of the screen, to be saved on files along with some text describing the experiment;
- a timer: in order to time experiments a timer can be started (stopped, cleared) using the mouse. It can also be controlled by other agents, via writing in a specific context.
- a friendly management of the screen, with scrolling, resizing etc.

Implementation

The Performance Monitoring System includes a number of agents, depending on the number of machines involved in the current session. Three types of agents compose the system:

1. *fperf*: keeps the information on the screen up to date, based on the data it receives from the other agents;
2. *mouse*: (created by the fperf agent) interacts with the user, e.g. catching X events and forwarding them to the *fperf* agent;
3. *perf*: periodically collects the statistics for the agents running on its own machine and sends them to the *fperf* agent for display.

One *fperf* and one *mouse* agent are needed, along with an instance of *perf* for each one of the machines involved. Any one of the agents could be replicated, even though replicating *mouse* could make things slightly confusing...

The agents communicate via shared data structures. When the *mouse* or the *perf* agent has new information to send to the *fperf* agent, it writes it in a context, each of them in a separate context. The *fperf* agent sets events on these two contexts, and whenever somebody writes into them it receives an activation and updates the display accordingly.

8.3. Pattern Matching

The basic mean for exercising control in Agora is the *event*: every time a context if modified an event can be generated. Every synchronization primitive can potentially be expressed in terms of events, but this may lead to inefficient scheduling of resources in some cases. For example, it may be necessary to maintain some form of global state information to activate expensive computations (e.g. searches) only when it is most profitable. In Agora, complicated scheduling algorithms can exploit user-level pattern-matching modules that filter the triggering of expensive computations: the Pattern Matcher is therefore the agent that sets the events, receives the activations, and allocates the available processor(s) to the most promising agent(s) e.g. posting activations to them.

This form of control + scheduling is derived from BlackBoard Systems in AI, and it has been described as *opportunistic scheduling*: the system applies its computing effort (focus of attention) to the most promising parts of the problem, applying problem-specific heuristics in the decision process. The amount of knowledge put in the scheduler could very well make it the most complicated component of

the system, to the point that some systems recursively apply the BlackBoard paradigm to the scheduling problem itself [28].

Since the kind of knowledge put in the scheduler is also very much domain-specific, Agora only provides a standard way to interface the user-provided Pattern Matcher, and a simple example of structuring of this type of agents. The interface consists of two remote procedure calls, and a standard activation type. Their C binding is shown in Table 8-1.

```
/* set_pattern(1)
 * Installs a new pattern for an agent.
 * Returns -1 in case of error.
 */
int set_pattern ( agent, act_type, mode, pattern)
int agent;
int act_type;
int mode;
char * pattern;

/* unset_pattern(1)
 * Removes all the patterns associated with the given
 * activation type for an agent.
 * Returns the number of patterns actually removed,
 * or -1 if an error occurred.
 */
int unset_pattern ( agent, act_type)
int agent;
int act_type;

/* Activation type to tell the Pattern Matcher
 * to re-evaluate the patterns
 */
#define PM_EVAL 2
```

Table 8-1: Pattern Matcher Interface

Invoking these functions from a user agent results into a request posted to the Pattern Matcher (if any) to install, remove, or re-evaluate the patterns. Usually, the Pattern Matcher itself will set events in the relevant contexts, with activation type PM_EVAL.

The example Pattern Matcher that is provided by Agora mimics Agora itself: patterns are simply events, e.g. they mention a context and the triggering mode as in set_event. The suggested approach, which may or may not be appropriate in all cases, is to pre-compile the pattern in some more efficient representation that allows a fast and possibly incremental evaluation, e.g. as the RETE algorithm [19] does for OPS5 rules. This approach is advisable when patterns do not change frequently during the computation, and when there are many and possibly complicated patterns involved.

8.4. Debugging

The Agora debugger is in experimental state, and it is not yet released to users. Since it is a very important tool for Agora, we will describe it along with the other tools that are already available. A more detailed description of the Agora debugger can be found in [20].

A parallel application poses control problems far more complicated than sequential applications do, both in programming it and in monitoring its execution. It is more difficult both to understand what the program is doing, and to be reasonably certain that the program is error free. It may become an arcane art to discover the causes of the error since debuggers often change the behavior of the user program to some extent causing many failure modes of the program to be undetectable during debugging. Moreover, a special kind of problem in parallel systems is caused by errors that manifest themselves only under certain conditions that are rarely met; when they do appear the best approach is to reproduce the exact sequence of events that lead to the error. We call this exact reproduction of events a *replay*.

At runtime, Agora collects enough information to allow an exact replay of the execution for debugging purposes. The user is able to exactly repeat, any number of times and at a human speed, *the* execution that failed. With such a mechanism, everything becomes detectable, since the user has both a time-scale and enough tools available to find *what is happening* in the system. The runtime information gathering mechanisms are sufficiently inexpensive that they are permanently part of the machine, eliminating any difference between debugging and production mode. Replay simplifies the debugging strategies to this very trivial one: *execute the program until it crashes, then look for the problem.*

The Agora debugger is a parallel program itself. It is decomposed into a main component that interfaces with the user and accesses the database, and a number of language-specific sequential debuggers, which have been minimally modified to allow the main component to control them. The sequential part of the debugging activity, e.g. debugging normal programming errors that only affect the internal state of an Agora agent, is therefore performed using a tool that is already familiar to the user, with minimal disruption of the user's habits.

Debugging Methodology

It is easy to make effective use of the debugger. The debugging cycle consists of the following steps:

- Agora checkpoints the shared memory before each experiments begins, and then proceeds to normal execution. The debugger does not need to be started at this time, it may be started later.
- If a module generates an exception during normal execution, the debugger is started, the system is rolled back to the beginning of the experiment and the execution replayed under control of the debugger.
- The debugger displays the evolution of crucial data structures to help localize the problem. Certain interactions between agents are also monitored as soon as evidence is gathered on who the culprit(s) is(are).
- Step by step execution of the culprit agent is performed to isolate the faulty code.

The faulty execution can be replayed as many times as necessary to fully understand the problem.

Debugging a distributed application can be viewed as a two level activity: debugging the system design, i.e. the system composition and the timing of the information exchanges between components and their volume and flow, and debugging the system parts, i.e. the content of this information as produced by some agent and the internal states of these agents during the computation. A wise debugging methodology in most cases is to first debug the system design using *stubs* for its parts, then to implement and debug each part in isolation and finally integrate the parts in the complete system and test it for time dependent bugs. The Agora debugger supports both levels of activity, although it does not *force* the user to follow any particular methodology.

Replay

In Agora, the only mean by which agents interact is through **activations**, context **writes** and context **reads**[7].

During the execution of any Agora application, each agent is periodically *checkpointed*. How often and when to checkpoint is specific of the application to which the agent belongs and it is the responsibility of the user: in a speech recognition system it would be "the beginning of an utterance", in a dataflow application "the input of a new set of data". Checkpointing is essentially a way of saving the *local state* of the agent, and making sure that it will be possible to bring it back to that state. Logically, we want to save on disk a complete memory image of the agent, and this could be a space and time consuming activity; in practice saving is incremental, only those memory pages that have actually changed during the execution are saved on disk.

Note that saving local state is obviously only part of the solution: one still has to guarantee that the *global state* of the system is and will be consistent with the saved local state. This is accomplished in two ways: using a *write-once* memory management policy, and using *version numbers*.

Agora's memory is managed in a write-once fashion: elements can be added, but not removed or modified. In this way, direct pointers to large structures can be given to the user's process, avoiding copies and extra locking. This also means that there is no way that a user can delete information from memory, and that nothing else needs to be done to have a history log for it. The only requirement is to stop the garbage collection process that would otherwise recover unreferenced memory, and turn the write operation into a null operation during replay.

There are, however, shared memory entities that do change: maps. In this case it would be definitely impractical to use a write-once approach, and therefore a different solution is used. Every map has a counter associated with it, which is incremented on each read *or* write operation. During execution, each agent logs in a private file the value of the counter retrieved by each read or write operation. During replay, the operation checks in the log file the value that the counter *should* have, and only proceeds when the counter reaches it.

---

[7]Nothing prevents a programmer from making use of other interprocess facilities, e.g. Mach IPC. In such a case, however, there is no guarantee that the replay mechanism will still work. More specifically, if the flow of execution depends on these other mechanisms, or if these maintain external state in files, the behavior during replay could differ.

8.5. Data Storage

In order to run experiments and exchange data, it must be possible to save and/or restore the runtime data structures on disk files. This is accomplished in Agora via a simple user-level agent. The representation in files must be machine independent, so that heterogeneous machines can access the same data files on central file servers. This avoids the waste of space and time in duplicating and converting files, which is substantial when large databases are involved, e.g. in speech recognition or vision research.

Machine independent file formats have usually used ASCII representations, given that hardware designers will hardly ever agree on byte ordering issues, or floating point formats... Using ASCII translation is both inefficient and wasteful, therefore Agora uses a direct binary representation. Each machine stores data in its natural byte order and floating point format. Agora then uses the information about the *type* of the data structure to convert a file if it has been produced by an incompatible architecture.

The format of the file is as follows:

```
(deftype FileFormat
  (int8 byte-order)
  (int8 padding 3)
  (type-descriptor type)
  (int8 data 0))
```

The first byte indicates the machine architecture that produced the file, with bits to identify the byte order and float representation used. Note that at least the **byte** seems to be an agreed-upon entity. Three padding bytes are added to compensate for alignment problems, and for possible extensions. Based on the byte order information, the *type descriptor* (see Chapter 2) can be read-in: this is described by a standard descriptor (the *meta* type descriptor), and if necessary it can be converted (see 6.2 for details about the translation algorithm). At this point, all the necessary information has been recovered, and the file-type can be checked against the type of the data structure. If they match, as they should, the rest of the file can be read-in, converted if necessary, and put in the data structure. The converse process of saving a data structure is simpler: the processor identifies itself in the first byte, and after the padding it simply dumps a binary image of the type descriptor and of the content of the data structure.

8.6. Loading of Agent Images

Mach currently provides shared memory facilities among processes in a hierarchical fashion: memory is allocated in a process, and only descendent processes may inherit it. Memory cannot be allocated and shared with existing processes, and the system provided loader e.g. the Unix **exec** system call, deallocates all the process' memory and allocates it anew. We have found ways of coping with the first limitation (see Section 5.3), and we had to find ways to overcome the second as well. This section describes the Agora loader, a user-level substitute for the exec system call that allows us to execute different images in our agents and preserve shared memory.

What the Loader does

The loader has exactly the same semantics that Unix's *exec* has: replace the current process' executable image with a different one, and transfer control to a pre-specified location in the new executable image. It is usually invoked shortly after the new process has been created via *fork*. In Agora this happens inside the **new_agent** primitive when invoked with the request of creating the new agent from a different executable image. In fact, the user is totally unaware of the loader in his programs: he only needs to know about new_agent.

The loader is physically a standard Unix program, which is invoked with a request to load an initial agent, e.g. the AgoraShell or the Agora Lisp startup program. Looking at the information contained in the executable image, it allocates enough memory to hold the text and data portions, it reads them in, and transfers control to the entry point specified in the executable image. When invoked again from new_agent, it deallocates most of the memory owned by the current process, and repeats the above procedure. Care is taken so that the code of the loader is accessible by any new agent. In this way the whole procedure is repeatable: every new agent can invoke new_agent which forks a new process and invokes the loader again etc.

The Case for Lisp

CMU CommonLisp is physically composed of two parts: a small startup program that is invoked from the Unix shell, and the Lisp corefile proper that holds all the interpreter and pre-loaded code. New corefiles can be generated by users that want to have their code already loaded when they enter Lisp, and indeed in Agora we use this facility.

Our initial approach was to incorporate the startup code inside the loader, e.g. providing a special executable name for Lisp agents. This was a dreadful solution, which lead us in all sorts of trouble. The solution we use now is simpler: the Lisp startup program has been turned into an Agora agent, and therefore it works exactly like the normal version. No special code in Agora is needed to handle Lisp any more.

Problems

Since the loader must be visible in the user's name space, a special linking procedure is needed. This makes Agora agents different from normal Unix executables, since they cannot be started with Unix exec even if it saved shared memory. Moreover, this can cause name conflicts in the linking phase. While a single identifier needs to be exported, i.e. the *load* function that replaces (exec), all the global identifiers in the loader's executable image become unnecessarily visible.

The loader is the most machine-specific tool Agora has: it needs to know about both the memory layouts of Unix/Mach processes, and about the structure of executable files on Unix/Mach. These are different among the various machines. For example, the RT PC uses separate text and data segments while the other machines do not. The Encore Multimax represents executable files in a format (the System-V COFF format) that differs from the way the other machines do (e.g. in Unix BSD exec format).

For these reasons, the loader has been the most painful part of the system, both in coding it, in porting it, and in maintaining it. We look forward to fixes to Mach that will make this tool unnecessary, and we do not advise anybody to follow this road again!

9. Example

This chapter contains two examples. The first example illustrates how the definition of a single shared data structure suffices to solve a non-trivial problem such as creation of multiple name spaces. Other parallel schemes, e.g. messages, would not solve it with the same simplicity, or with the same generality. The second example shows how Agora has been used to create the prototype of a distributed *make-like* facility. It also shows how *CFrames* has been used in prototyping components of an integrated programming environment.

9.1. A Name Service

When two parallel components must communicate with each other, they must be able to *name* each other, or some common entity. Some parallel languages solve this problem via scoping rules (e.g. CSP, Ada). This is not possible if there is no overlap between the two processes. This happens, for instance, when the two processes are written in separate languages. Hewitt [2] has formalized this visibility problem for object systems, and solved it by postulating the existence of a *mail service.*

An important case in point are message based systems: in order to exchange messages between them, two processes must exchange some form of handle (port, mailbox, channel ..). This is often done through a third party: a universally known process that acts as a "database" of handles. Other processes can request that a handle be added to the database or retrieve a handle by using a string as the key. In Agora the problem has a very simple solution: a shared structure that maps strings to handles. This structure is described in Table 9-1 as an abstract data type which has three defined operations: insertion of a pair [name, handle], removal of a pair and retrieval of the handle given the name.

Note the implementation of the *check-in* operation: since the context might be updated between the read and write operations, we wrap the non-parallel *seq-checkin* function inside a call to *cx-atomic-execute.*

The data structure defined in Table 9-1 could be used to pass some meaningful integer value between agents. This is illustrated in the code fragment of Table 9-2, where a C and a Lisp agent exchange a context identifier in this way.

Table 9-7 is an example of a simple C program that uses the name service: a utility agent that logs on a terminal all the traffic of names in and out of the table. To do so, the agent sets up an event on the name service context, asking to be notified whenever it changes. Every time a name is checked in or out, an activation for this agent is generated. The activation contains an indication of which particular name changed. This example also shows the model structure of an agent: the agent waits for events to fire up and simply dispatches to the relative procedures. Activation type 0 is a standard way of asking an agent to (re)initialize itself: in our case the code is trivial and has been expanded in line.

There is one last thing to note about this example: there is a bootstrap problem. How does an agent know about the manifest value NAMESERVICE ? The pseudo-constant NAMESERVICE must be substituted by a real value. There are two solutions for this: either pass a value in the command line to the agents, or use a standard, pre-defined value. Given the convenience of having a standard name service, Agora automatically creates one at startup time, which is then used internally. Applications

```
;; User-level NameService
;;
;; Operations that a NameService provides are
;;       check-in         a name,object pair
;;       lookup           an object
;;       check-out        a name
;; These are described as custom functions of the
;; "NAME-SERVICE" context
;;
;; The context is a simple one, hashed, and since we do not expect
;; too many name clashes, the hash table size is small.
;; The objects that we associate to names are simple integers.
;;
(deftype handle
    (int32 value))

(defcontext name-service :hash (25)
    ((data-ref value handle)))
;;
;; *- Check-In *-
;;
(defaccess checkin (ns name entity)
  (declare
    (fixnum ns entity)
    (string name))
  (cx-atomic-execute ns
    (seq-checkin (ns name entity))))

(defun seq-checkin (ns name entity)
  (declare
    (fixnum ns entity)
    (string name))
  (let ((a (name-service-read ns name)))
    (if
      (and a
        (/= -1
          (handle-value
            (name-service-value a))))
      nil        ; already in
      (name-service-write ns name entity))))
;;
;; *- Check-Out *-
;;
(defaccess checkout (ns name)
  (declare
    (fixnum ns)
    (string name))
  (name-service-write ns name -1))
;;
;; *- Look-Up *-
;;
(defaccess lookup (ns name)
  (declare
    (fixnum ns)
    (string name))
  (let
    ((a (name-service-read ns name)))
    (if a
        (handle-value
          (name-service-value a))
        -1)))
```

Table 9-1: An Example Shared Data Type: the NameService

create their own name services and use the standard one only for bootstrapping.

```
agent1()                /* In the C language */
{
        ....
        a = cx_create(..);
        checkin( NAMESERVICE, "our-private-context", a);
        ....
}

(agent2 (....)          ;; in CommonLisp
  (let ((b (lookup NAMESERVICE "our-private-context")))
     ....))
```

Table 9-2: Use of the NameService from Multiple Languages

Given the description of Table 9-1 in the specification language, the Agora compiler generates the code for the C language shown in Tables 9-8 and 9-9. The compiler also generates the code for the Lisp language shown in Table 9-10 which makes use of the CMU-CommonLisp foreign-function facility. Some code, namely the translation of the access functions, has been omitted to avoid repetitions. The C version of the function *name-service-create*, which is also synthesized by the compiler, is analogous to the Lisp version and is not shown.

9.2. A Distributed Make

In a large workstation environment machines are idle for about 80% of the time [33]. One of the ways in which those resources could be used is in distributing the compilation and linking loads of a busy machine over the network. With Agora, we built a prototype for a utility that performs this distribution. This *distributed-make* facility, given enough information about the user's application and about the idle machines, starts the recompilations and linking activities as much as possible in parallel, and on separate machines. Since it was only built for demonstrational purposes, this section only provides information about the overall structure and some details about specific problems.

Structure

A main Lisp agent (called *Pmake* in the following) starts and controls the compilations, and other agents perform individual tasks upon request of the main agent. The goal of the utility is to build a *system*, defined as a hierarchy of agents and systems. Each agent requires a number of source files (in some programming language), and possibly other link-time program libraries. On Unix, this information is provided to *make* by the user in a file, called *makefile*. Our prototype used instead a graphical user interface as the front-end to an object database [7], but it is certainly possible to automatically translate the information contained in a makefile and insert it into the database used by Pmake.

CFrames is used in the prototype to implement the object system. As a reminder, these are the Cframes routines used in the following pieces of code:

avalue: adds a value to a slot in a frame;

g1value: returns the first value of a slot in a frame;

gvalues: returns the list of all values of a slot in a frame.

Consider the case where the executable image for an agent written in C must be (re)constructed for a certain machine. *Pmake* first invokes a function to check and possibly recompile the existing object files for that machine, e.g. it controls whether the sources have been changed. This check relies on the date of the files. It is easy for Lisp agents to decide if they are up to date or not, but it is much less obvious for C agents, since C allows implicit dependencies via the *include* pre-processor command. Modifying the content of the included files potentially makes the compiled code of a source file out of date, even if it has a newer timestamp than its source. C also presents other problems because the preprocessor could find the included files in different places, depending on where it is invoked from and/or the status of the user's search paths.

To realize this checking *Pmake* uses a language specific tool, called *scan*: this is a C agent that scans a C source file looking for the included files. While doing this, it keeps the *Cframes* database up to date, making the appropriate changes. *scan* agents do not interfere with each other when scanning different source files, therefore this first task can be done in parallel for all the agents, and for all the sources of each agent. The code in Table 9-3 illustrates how the agent operates. The list of directories where the included files can be located is a separate piece of information, which is also kept in the database.

```
...
while (! end-of-file)
{
    read next valid pre-processor command
    if (strcmp (command, "include") == 0)
    {
        find the name of the included file
        avalue (source, "INCLUDING", STRING, name);
   /* 'source' is the frame describing the source file */

    }
}
```

Table 9-3: How *Scan* Operates

The fragments of Lisp code in Table 9-4 show how *scan* is used by the Pmake agent.

Once all the scanning is done, Pmake collects the results and proceeds to the next phase. It is easy now to compare the date of the sources and included files with the date of the object. If a source file must be compiled, another routine, 'compile-source' is invoked (see Table 9-5): this function invokes another agent, *tool*, to accomplish the compilation phase. As for the *scan* agent, any number of *tool* agents can be running in parallel, so that compilations will run in parallel too. The *tool* agent is a general purpose one: its goal is to invoke a standard Unix program and wait for its termination. It is then again *tool* which is invoked for the final linking phase, when all compilations are successfully completed.

The 'tool' agent is a general purpose agent, used to execute Unix tools and wait for their termination. Table 9-6 illustrates how the agent operates (minor details have been omitted for clarity).

```
(defun check-C-dependencies (agent machine-type)
  (look-for-all-C-dependencies agent "sources" machine-type)
...
)

(defun look-for-all-C-dependencies (frame relation machine-type)

  ;; 'mark' marks the argument frame: in this
  ;; way we will not visit it twice

  (mark frame)
  (dolist (file (gvalues frame relation))
    (if (> (glvalue file "lastupdate") (glvalue file "lastscan"))

  ;; The file needs to be re-scanned.
      (invoke-tool "scan" machine-type file))

  ;; we must recur on the files included by the current frame
  ;; if they have not been already marked

    (if (unmarked file)
      (look-for-all-C-dependencies file "INCLUDING" machine-type)))
...
)

(defun invoke-tool (tool-name machine-type &rest args)

  ;; 'check-running-agent' checks if an instance of the agent
  ;; 'tool-name' is running on a machine of type 'machine-type':
  ;; if it is not found,  new-agent is invoke to start
  ;; 'tool-name' with 'machine-type' as command line

  (check-running-agent tool-name machine-type)

  ;; We do not want to immediately wait for the completion of
  ;; the command sent to 'tool-name', so we use the remote
  ;; procedure calling functions  a-snd here and a-rcv later,
  ;; instead of a-rpc which does both.

  (a-snd (asm-lookup (strcat tool-name machine-type)) args *MY-CX*)

  ;; -   'asm-lookup' looks up the context with the NameService
  ;; -   '*MY-CX*' is a global variable which holds the context where
  ;;                this agent receives replies;
)
```

Table 9-4: How *Scan* Is Used

```
(defun compile-source (source-frame machine-type)
...
  ;; 'cflags' and  'file-path' are extracted
  ;; from the information contained in 'source-frame'.
  ;; 'cflags' contains a list of "-I" options to specify
  ;; where the included files must be located

  (invoke-tool "tool" machine-type "cc" "-c" cflags file-path)
...
)
```

Table 9-5: Compilation Of C Source Files

```
int check_for_exec (file)
  char *file;
{   /* check if file is executable or not       */
  return (access (file, X_OK));
}

int tool_go (params)
  char **params;
{
    /* 'params' contains the parameter list.                       */
    /* The first parameter is the name of the Unix tool that       */
    /* is executed and the corresponding executable is located     */
    /* along the PATH Unix environment variable.                   */

    if (searchp (getenv("PATH"), params[0], file_path, check_for_exec) != 0) {
        fprintf (stderr,"Tool could not find file %s on PATH\n", param[0]);
        return (ENOENT);
    } else
        *v++ = file_path;
   /* ... collect all the remaining arguments in the vector v,    */
   /* then start the Unix program                                 */
   if (!(pid = fork ())) {
       execve (v [0], v, environ);     /* child, will never return */
   }
   while (pid != wait (&status))
     ;
   return (status);
}

tool_main (argc, argv)
  int argc;
  char **argv;
{
  CONTEXT tool_cx;
  int result;

   /* this is the structure received whenever an rpc  */
   /* activation is sent to us:                       */

  struct {
          int size;
          CONTEXT reply;
          char params[1]; /* varsized */
        } *rpc_params;

  agora_init ();
  wait_activation( BLOCK, 0);

   /* create a CONTEXT for input .. */
  tool_cx = tool_create ("tool_call");

   /* ... and declare it be used for rpc, as well as */
   /* checking it in with the Name Server.  Our name */
   /* and the type of the machine we run on make up  */
   /* the name that is checked in                    */

  asm_checkin (strcat (argv [0], argv [1]), rpc_on (tool_cx));

   /* wait for invocations: */
  while ((rpc_params = a_rcv (tool_cx, BLOCK)) != NULL) {
     result = tool_go (rpc_params->params);
     /* reply to the rpc: */
     a_snd (&result, rpc_params->reply);
  }

   /* we got a bad request: let's terminate */
  agora_finished (EBADREP);
}
```

Table 9-6: The *Tool* Agent

```
#include <stdio.h>
#include <asm.h>

log_print()
{
        int             value, the_context;
        char           *address;

                        /* which name changed ? */
        get_activation( &the_context, &address);
        printf("Name %s has been checked ", address);
                        /* see what happened */
        if ((value = lookup(the_context, address)) < 0)
                printf("out.\n");
        else
                printf("in, value %d.\n", value);
}

log_main( argc, argv)
char **argv;
{
        int             running = 1;

        agora_init();

        while (running)
                /* Block until we are activated */
                switch (wait_activation(BLOCK, ANY_ACTIVATION)) {
                case 0:
                        /* Initialization: set up events */
                        set_event(self(unique_id), 1, NAMESERVICE,
                                  T_CONTEXT, EVENT_CHANGES);
                        break;
                case 1:
                        /* An event fired */
                        log_print();
                        break;
                default:
                        /*
                         * Either we're asked to terminate, or
                         * we don't know how to handle this
                         */
                        running = 0;
                }
        agora_finished(0);
}
```

Table 9-7: Example: Logging of NameService Operations

```
/*   File created by ATG (Agora Type Generator) version  1.22    */

#include <asm.h>

typedef struct {
        int      value;
} handle;

typedef struct {
        handle  *value; /* map entries */
} name_service;

typedef struct {
        int value; /* init values */
} info_name_service;
```

Table 9-8: C Header file *ns.h*

```
/*   File created by ATG (Agora Type Generator) version  1.22    */

#include "ns.h"

/*  function name_service_create */
.....

name_service *checkin (ns, name, entity)                    /* access function */
int ns, entity;
char *name;
{
        return (name_service *)
            cx_atomic_execute(ns, seq-checkin, ns, name, entity);
}

static
  name_service *seq-checkin (ns, name, entity)              /* access function */
int ns, entity;
char *name;
{
        name_service   *a;

        a = (name_service *)
                cx_read( ns, name);
        if (a && (-1 != a->value->value))
                return 0;
        else
                return (name_service *)
                        cx_write(ns, name, &entity);
}

name_service *checkout (ns, name)                           /* access function */
int ns;
char *name;
{
        int t;

        t = -1;
        return (name_service *)
                cx_write( ns, name, &t);
}

lookup (ns, name)                                           /* access function */
int ns;
char *name;
{
        name_service   *a = (name_service*)
                cx_read( ns, name);

        return (a ? a->value->value : -1);
}
```

Table 9-9: C Code file *ns.c*

```
;;;   File created by ATG (Agora Type Generator) version  1.22

(use-package '("LISP" "SYSTEM" "EXTENSIONS"))

(def-c-record handle
        (value  int))

(def-c-pointer **handle *handle)

(def-c-record name-service
        (value  *handle))

(def-c-record info-name-service
        (value  int))

(def-c-routine ("cx_read" name-service-read) (*name-service)
        (context        int)
        (address        string))

(def-c-routine ("cx_write" name-service-write) (*name-service)
        (context        int)
        (address        string)
        (list   *name-service))

(def-c-routine ("cx_create" name-service-create0) (int)
        (typename       asm::c-name)
        (info *info-name-service)
        (table-size     int))

;;; Creation function for name-service contexts

(defun name-service-create (&optional info &key table-size)
  (declare (type list info))
  (unless table-size
    (setq table-size 25))

  (let ((m0 asm::*MAP-ANY*)
        c1
        (a0 (make-info-name-service))
        ret)    ;;; returned value and temporaries

        ;; initial phase: the context name-service must be
        ;; created based on the info parameter

    (setf (alien-access (info-name-service-value a0))
        (if (setf ret (nth 0 info)) ret (coll-create "handle")))
    (setf c1 (alien-access (info-name-service-value a0)))
    (setf m0
        (asm:cx-top-map
            (setf ret (name-service-create0 "name-service"
                (lisp::alien-value-sap a0) table-size))))

        ;; type checking phase

    (cond (info
           (cond ((or (string/= (asm:element-type c1) "handle")
                      (string/= (asm:map-type m0) "name-service"))
                  (error "Type checking error")))))
    ret))

;; the code of the access functions remains largely unchanged
......
```

Table 9-10: Lisp Code file *ns.lisp*

10. Conclusions

The Agora Project's goal is to facilitate the development of parallel, multiple language applications by providing a friendly development environment and a simple and efficient execution model. Some of the ideas that are keystones for the project include:

- *shared memory is a viable communication abstraction* between components implemented in different languages;
- *a structured shared memory can be implemented* on non-shared memory architectures and across heterogeneous machines;
- *event-driven execution is a convenient control mechanism* for a shared memory model.

Agora is very flexible as far as **control** is concerned: both *Data Driven* and *Control Driven* designs are possible. The basic event-driven mechanisms encourage data driven designs, and Remote Procedure Calls are available as a simple package built on top of the standard Agora abstractions. It is also simple to use the basic Agora primitives to build Dataflow-like systems. A BlackBoard System is another control scheme that is closely related to Agora. In this respect, Agora's shared memory acts as a central blackboard where all the modules can browse and add/modify information. A module can declare itself interested in every change to certain specific data structures. Whenever those are modified, the module is notified through an activation, a piece of information that is queued to a per-module queue and tells which specific components of the data structure had been changed. Unlike BlackBoards, there is no central scheduler to filter and schedule these activations. A user can provide his/her own scheduler, since module activations can be controlled at the single data structure access level. Complicated scheduling algorithms can exploit pattern-matching modules that filter the triggering of (potentially) expensive computations.

The largest gain in decomposing an application in a BlackBoard-like fashion is **modularity**: components of various complexity and size can be easily merged in a manner totally independent of their internal flow of control. The rules to make them fit together are as simple as type-checking the common data structure definitions, and can be negotiated in advance and reinforced using the Data Type Generator tool. Given that no specific flow-of-control restriction is imposed by the system, an application can be decomposed in modules in the most (functionally) appropriate way, encouraging clean and understandable designs.

A major goal for Agora is **efficiency**. This is both in the sense of efficient execution of applications, and in the sense of their efficient development. Efficient execution is obtained avoiding redundant copy operations (shared data structures are directly mapped in the address space of the various processes), doing garbage-collection of reusable storage incrementally and in parallel, and buffering expensive network communications (Agora packets are seldom smaller than the maximum size).

Efficient program development is obtained extending into the parallel world the environments that are most cherished by programmers: **Unix and CommonLisp**. The primary interface to Agora can indifferently be an extended version of the C-Shell, or the standard CMU-CommonLisp environment. Both offer the ability to quickly develop and interpretively test modules that use the Agora primitives. Besides the standard Unix/Lisp tools, Agora offers other tools that simplify the labor intensive task of

debugging and tuning an application: a parallel and distributed multilanguage debugger capable of replay of executions, and a performance monitor that graphically pictures the evolution of a system during the execution. In this way the performance bottlenecks are easily identified.

Agora also offers **dynamic configurability**: unlike many other parallel systems, components may be started at *any* time, allowing the system to tailor its computing resources to the dynamic needs of the application. No "system reconfiguration" is ever necessary when experimenting with Agora. Interpretive techniques, an extensive toolset, and flexibility are the keys to achieve efficiency, in the sense that has often been indicated as *rapid prototyping*.

Agora is distributed together with the Mach operating system release. It currently runs on DEC Vax, IBM RT PC, Sun, Encore Multimax and all possible combinations of these machines. The languages currently supported are C, C++ and CommonLisp. A first version of Agora has been operational since September 1986 and is used in the development of various applications. A detailed evaluation of the performance can be found in [8].

References

[1] Adams, D. and Bisiani, R.
The Carnegie-Mellon University Distributed Speech Recognition System.
Speech Technology 3(2), April, 1986.

[2] Agha, G. and Hewitt, C.
Concurrent Programming Using Actors.
Object-Oriented Concurrent Programming.
MIT Press, Cambridge, MA, 1987.

[3] Bauer, M.M. and Lerner, R.A.
CFrames: A Miniature Frame System.
August, 1987.
Agora Working Memo.

[4] Baron, R., Rashid, R., Siegel, E., Tevanian, A., and Young, M.
Mach-1: An Operating System Environment for Large Scale Multiprocessor Applications.
IEEE Software Special Issue, July, 1985.

[5] Betz, D.M.
XLISP: An Experimental Object-oriented Language
1.6 edition, , Manchester, NH, 1986.

[6] Bisiani, R., Alleva, F., Correrini, F., Forin, A., Lecouat, F., Lerner, R.
Heterogeneous Parallel Processing, The Agora Shared Memory.
Tech. Report CMU-CS-87-112, Carnegie-Mellon University, Comp. Science Dept., March, 1987.

[7] Bisiani, R., Alleva, F., Correrini, F., Forin, A., Lecouat, F., Lerner, R.
Heterogeneous Parallel Processing, The Agora Programming Environment.
Tech. Report CMU-CS-87-113, Carnegie-Mellon University, Comp. Science Dept., March, 1987.

[8] Bisiani, R. and Forin, A.
Architectural Support for Multilanguage Parallel Programming on Heterogeneous Systems.
In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 21-30. IEEE, Palo Alto , October, 1987.

[9] Bisiani, R., Lecouat, F. and Ambriola, V.
A Planner for the Automation of Programming Environment Tasks.
In *Hawaii International Conference on System Sciences (HICSS-21).* Kona, HA, January, 1988.

[10] Bobrow,D.G. and Stefik,M.J.
A Virtual Machine for Experiments in Knowledge Representation.
Technical Report, Xerox Palo Alto Research Center, April, 1982.

[11] Bobrow D.G., et al.
CommonLoops: Merging Lisp and Object-Oriented Programming.
In *Proceedings of OOPSLA'86, Sigplan Notices Vol.21 Nov 86*, pages 17-30. Portland, Oregon, September, 1986.

[12] Brooks, R.A. et al..
Design of an Optimizing Dynamically Retargetable Compiler.
In *Symposium on Lisp and Functional Programming*, pages 67-85. ACM, June, 1986.

[13] Cheriton D.R.
The V Kernel: A Software Base for Distributed Systems.
IEEE Software 1(2):26-33, April, 1984.

[14] Cheriton D.R.
Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design.
In *Proc. of the 6th Intl. Conf. on Distributed Computing Systems*, pages 190-197. May, 1986.

[15] Corkill, D., Gallagher, K., Murray, K.
GBB: A generic blackboard development system.
In *Proc. Nat. Conf. on Artificial Intelligence*, pages 1008-1014. AAAI, Philadelphia, PA, August, 1986.

[16] DoD.
The Ada Programming Language.
MIL-STD-1815A, 1980.

[17] Erman, L., Hayes-Roth, F., Lesser, V., Reddy, R.
The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty.
Computing Surveys 12(2):213-253, 1980.

[18] Foderaro, J.K., Sklower, K.L., Layer, K.
The FRANZ LISP Manual
UC, Berkeley, CA, 1983.

[19] Forgy, C.L.
Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.
Artificial Intelligence 19:17-37, September, 1982.

[20] Forin, Alessandro.
Debugging in a Distributed Environment with Artificial Intelligence Techniques.
PhD thesis, Universita' degli Studi di Padova, Padua Italy, September, 1987.
In Italian.

[21] Forin.A, Correrini.F.
Agora User's Manual
1.0 edition, Carnegie Mellon University, Pittsburgh, PA, 1988.

[22] Fox, M.S., McDermott, J.
The Role of Databases in Knowledge-Based Systems.
Technical Report, Robotics Institute, Carnegie-Mellon University, 1986.

[23] Gabriel, R.P., and McCarthy, J.
Queue-based multiprocessing Lisp.
In *Symposium on Lisp and Functional Programming.* ACM, August, 1984.

[24] Gelernter, D., et al.
Parallel Programming in Linda.
In *Proc. Int. Conf. on Parallel Processing.* IEEE Computer Society, August, 1985.

[25] Gettys, J., Newman, R., Della Fera, T.
Xlib - C Language X Interface Protocol Version 10.
Technical Report , MIT-Project Athena, January, 1986.

[26] Goldberg, A.
Smalltalk-80: the interactive programming environment.
Addison-Wesley Publishing Co., 1984.

[27] Halstead, R.
Multilisp: A Language for Concurrent Symbolic Computations.
ACM Transactions on Prog. Languages and Systems , October, 1985.

[28] Hayes-Roth, B.
Blackboard Architecture for Control.
Journal of Artificial Intelligence 26:251-321, 1985.

[29] Hoare, C.A.R.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10), 1974.

[30] Kai Li and Paul Hudak.
Memory Coherence in Shared Virtual Memory Systems.
In *Proceedings of the Fifth Annual Symposium on Principles of Distributed Computing*, pages 229-239. ACM, 1986.

[31] Liskov, B., Guttag, J.
Abstraction and specification in program development.
McGraw-Hill, 1986.

[32] McDonald D. editor.
CMU Common Lisp User's Manual. Mach/IBM RT PC Edition.
Technical Report CMU-CS-87-156, Carnegie-Mellon University, September, 1987.

[33] Nichols, D.
Using Idle Workstations in a Shared Computing Environment.
In *11th ACM Symposium on Operating Systems Principles.* ACM, Austin, TX, November, 1987.

[34] Notkin, D.
The Gandalf Project.
The Journal of Systems and Software 5(2):91-105, May, 1985.

[35] Rashid, R.F.
An Interprocess Communication Facility for Unix.
Report , Carnegie-Mellon University, Comp. Science Dept., June, 1980.

[36] Seitz, C. L.
The Cosmic Cube.
Comm. ACM 28(1):22-33, January, 1985.

[37] Spector, A.Z., et al.
The Camelot Project.
Database Engineering 9(4), December, 1986.
Also available as Technical Report CMU-CS-86-166, Carnegie-Mellon University, November 1986.

[38] Stroustrup, B.
The C++ Programming Language.
Addison-Wesley Publishing Co., 1986.

[39] Sun Microsystems.
Sun Remote Procedure Call Specification.
Technical Report , Sun Microsystems Inc., 1984.

[40] Tevanian, A., et al.
Mach Threads and the Unix Kernel: The Battle for Control.
Technical Report CMU-CS-87-149, Carnegie-Mellon University, August, 1987.

[41] Vrsalovic, D., Segall, Z., Siewiorek, D., Russinovic, M.
MPC: Multiprocessor C
PIE Project, Carnegie-Mellon University, 1987.

[42] Wirth, N.
Programming in Modula-2.
Springer-Verlag, 1982.

[43] Young, S. R., Ward, W. H., Hauptmann, A. and Lee, Z.
An Integrated Speech and Natural Language Dialog System: Using Dialog Knowledge in Speech Recognition.
Computational Linguistics , 1988.
(Submitted).

Index

:hash 10
:linear 10

A_node 45
A_rcv 21, 23
A_roff 23
A_ron 23
A_rpc 19, 21, 23
A_snd 21, 23
Abstract Data Type 4
Access Function 4, 5, 9, 11
Activate 33
Activation 32, 33, 59, 75
Activation, initialization 20
Activation, Queue 34
Activation, tag 21
Activation, termination 20
Active Object 4
Addressing 9, 10, 12
Ag_err 22
Agent 4, 5, 9, 33
Agent Identifier 34
Agent_status 22
Agora 1, 36
Agora Shell 5, 31
Agora, package 26
Agora_finished 21
Agora_init 21
AgoraServer 5, 6, 37, 38
Aliens 27
Array, variable sized 10
ASM 5, 21, 38
Asm_init 21
Asm_statistics 19, 22
Ate 12
Atg 12
Audience 1

Basic Functions 11
Basic Types 10
BlackBoard System 4, 7, 56, 75
Bottleneck 42
Buffer_tune 36
Builtin Function 32

C 3, 19
C types 13
C++ 19
C-Shell 5, 31
C_count 35
Cframe 46
CFrames 33, 45
Client-Server 4
Coherence 37
Compilation 12
Compiler 4, 5, 9
Cond 12
Configuration 34, 35, 36, 76
Context 5, 9, 10, 37
Context-ref 10
Control-Driven 75
Copy-on-reference 37
Create 35
CSlot 46
Customized Environment 6
Cvalue 47
Cx_atomic_execute 11, 19, 23
Cx_create 11, 22
Cx_destroy 11, 22
Cx_read 11, 22, 59
Cx_share 19, 22
Cx_top_map 22
Cx_write 11, 19, 23, 53, 59
Cx_write_s 23

Data 10
Data definition 9
Data Translation 6, 40, 41, 60
Data Transport 38, 39
Data Type Generator 4, 9, 75
Data-Driven 34, 75
Data-ref 10
DataFlow 75
Debugger 5, 6, 53, 58
Debugging methodology 59
Def-c-record 13
Defaccess 11
Defcontext 10
Defstruct 13
Deftype 9
Defun 11
Distributed Application 6
Distributed Database 6
Distributed Make 65
Do 12

Efficiency 75
Enum32 10
Enumerated Types 10
Eval Server 26
Eval_events 19, 22
Event 34, 56
Event-Driven 4, 75
Exec 32, 60

Field 9
Find_agent 21
Fine-grain Parallelism 7
Float32 10
Float64 10
Floating point 42
Foreign-function 25
Fork 61

G1stvalue 47
Garbage Collection 5, 6, 27
GarbageCollector 53
Gc 35
Get_activation 22
Get_shared_mem_usage 22
Gnxtvalue 48
Granularity, control 7
Granularity, data 10

H_map_update 23
Heterogeneity 3, 4, 60

Include 9
Init_fountain 50
Int16 10
Int32 10
Int8 10
Invoke 50
IPC 39
Is-a 45
Iterator 16

Job control 32

Kill 32

L_map_update 23
Lagents 34, 54
Lcolls 35
Let 12
Light-Weight Processes 7
Lisp 3, 25
Lisp Agent 26
Lisp Shell 25, 36
Lisp startup 27, 61
Lisp Tools 26
Lisp types 13
Load 61
Loader 32, 53, 60
Login Shell 36
Login_parameters 35
Ltypes 35

Mach dependencies 39
Map 12
Map_bounds 12, 23
Map_create 23
Map_deref 23
Map_destroy 23
Map_find 12, 23
Map_gfree 23
Map_iterate 23
Map_next 12
Map_share 23
Map_underef 23
Map_update 12, 19, 53
Memory-based, models 37
Message-based, models 37
Messages 3, 37
Meta-Language 4, 5
Model 19, 26
Modularity 75
Monitors 4
Multiple inheritance 45

Name Service 63
Naming 40
New_agent 6, 19, 21, 32, 34, 61
New_machine 35
Number-crunching 26

Object Oriented Programming 4
Operating System 3, 37

Password 35
Pattern Matcher 34, 53, 56
Pattern Matching 34, 75
Performance Monitoring 5
PerformanceMonitor 53, 54
Permanent Storage 5, 6, 53, 60
Planning 6
PM_EVAL 57
Process migration 42
Programming Environment 6, 31, 75
Project Members 1
Put_activation 22

Rapid Prototyping 76
Read_element 35
Read_eval_print 50
Regulate 22
Remote operations 38
Rendez-vous 21
Replay 6, 58
Reusable Code 25, 31
RPC 3, 19, 21, 23, 25, 75
RTest 49

Scenario 3
Scheduling 75
Script 32
Security 42
Self 21
Semantic network 45
Set 12
Set_agent_name 22
Set_event 22, 34
Set_pattern 19, 23, 34, 57
Set_statistics 22
Sh_mem_usage 36
Share 35
Shared Data Type 4
Shared Memory 3, 37, 75
Shared Memory, networked 37
Shared_free 19, 24
Shared_malloc 19, 23
Shell 5, 32, 36

Shoot_agent 22
Sleep-wakeup 37
Statistics 34
String_ValueP 49
Struct 13
Stub Generator 3
Synchronization 7, 33

Terminate 21, 32, 33
Test-and-set 37
Tools 53
Type Descriptor 5, 13

Uint16 10
Uint32 10
Uint8 10
Unix 5, 31
Unix signal 37
Unset_event 22, 34
Unset_pattern 19, 23, 34, 57
User Interface 7
User's View 5

Wait_activation 19, 22, 34
With-locked-cx 16
Write_element 35

X 32, 54
Xleval 50
Xlisp 33, 45, 50