California Polytechnic State University, San Luis Obispo, CA 93407

DESIGN INSTITUTE REPORT
CADRU-10-95

Inter-Process Communication in Support of the ICDM Framework

Kym Jason Pohl	Computer Science Department
James Taylor	Intellicorp, Mountain View, CA
Leonard Myers	Computer Science Department
Jens Pohl	Architecture Department

June 1995

CAD RESEARCH CENTER
College of Architecture and Environmental Design

DESIGN INSTITUTE RESEARCH REPORT: CADRU-10-95

Inter-Process Communication in Support of the ICDM Framework

Kym Jason Pohl Computer Science Department
James Taylor Intellicorp, Mountain View,CA
Leonard Myers Computer Science Department
Jens Pohl Architecture Department

CAD Research Center

College of Architecture and Environmental Design
California Polytechnic State University, San Luis Obispo, CA 93407

Abstract

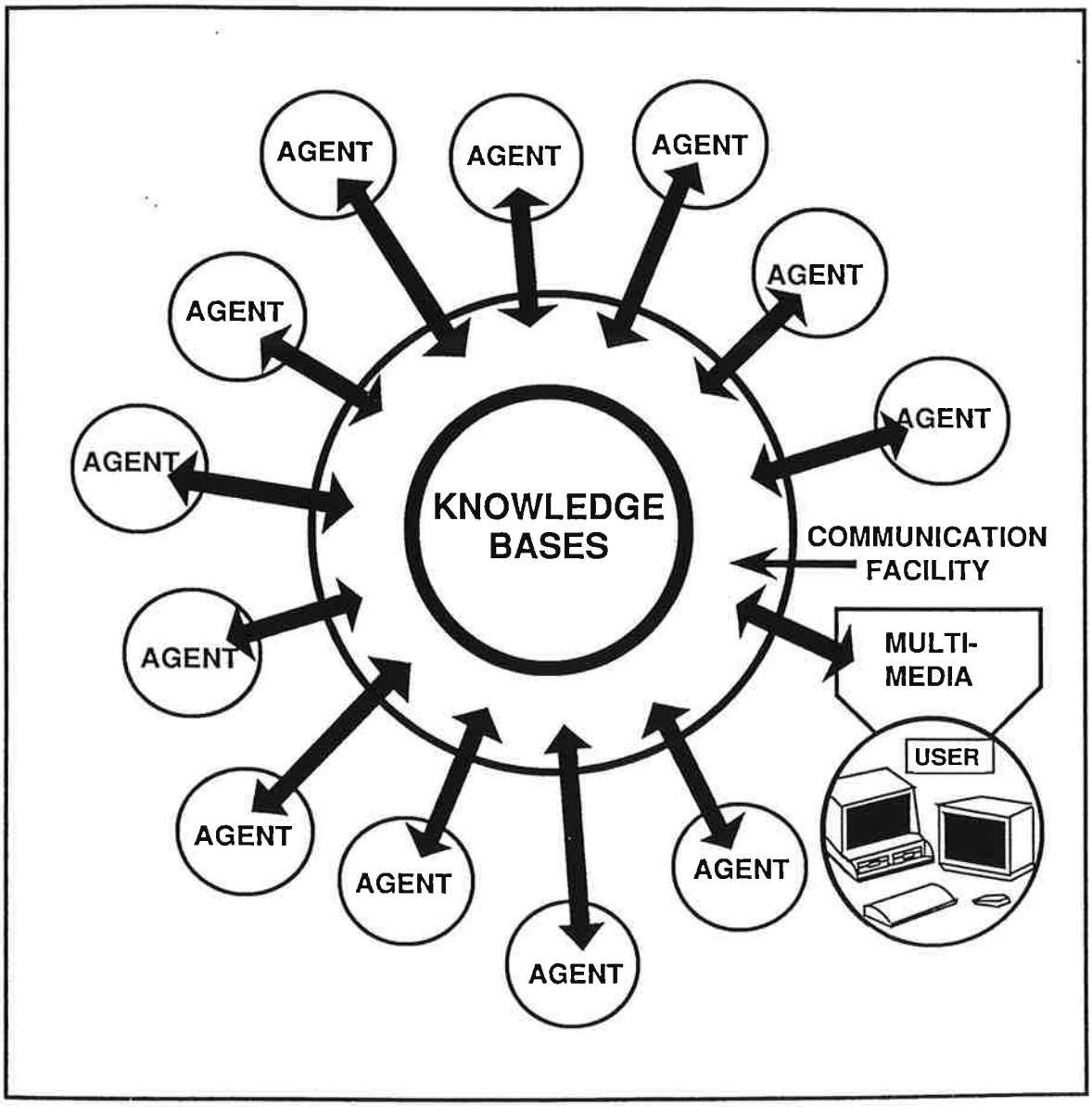
This report describes the progressive evolution of message-passing facilities that have been developed in the CAD Research Center in support of the various implementations of the ICDM (Intelligent Cooperative Decision Making) framework over the past decade.

Commencing with the multi-agent communication facility used in the first version of the ICADS (Intelligent Computer-Assisted Design System) prototype in 1989, the report traces the development of the Graphical X-Window Interface Builder (GXI), and the Mercury Message Management System (MMS), to the PVM-based Communication Management System (CMS) that currently forms part of the ICODES (Integrated Computerized Deployment System) and the FEAT (Force Employment Analysis Tool) under development for the US Transportation Command and the US Marine Corps, respectively.

It is clearly indicated in the report that with the increasing availability of commercial or public domain products, such as PVM (Parallel Virtual Machine), the CAD Research Center will be able to concentrate its efforts on the adaptation and extension of existing communication engines rather than the design and development of entirely new systems.

Keywords

agents, authorization, blackboard, communication, cooperative decision making, cooperation, coordination, distributed computing, event management, flexibility, message-passing, real-time, resource allocation.



Conceptual View of the ICDM Framework

Inter-Process Communication in Support of the ICDM Framework

1. Distributed Cooperative Decision-Support Systems

Since its formation in the mid-1980s the principal focus of the CAD Research Center has been the design and implementation of computer-based decision-support systems that can assist human decision makers in the solution of complex problems. The characteristics of such problems have been described elsewhere (Pohl et al. 1994) as including: many related issues; relationships that often appear to be more important than the variables that they connect; dynamic information changes; incomplete information and a high degree of uncertainty; and, changing solution objectives.

The approach that the CAD Research Center has followed is commonly referred to as distributed cooperative computing. In these systems several computers, usually but not necessarily workstations, are connected by a communication network. The application software consists of several processes that are distributed over one or more of the computers, and communicate with each other either directly or indirectly through some coordinating facility.

The concept of solving problems through the collaborative efforts of individuals within one group or several groups (i.e., the notion of a team) has become fundamental to most planning, design, management, and other decision making endeavors. It recognizes the parallel, rather than sequential, nature of information flows and the advantages of decentralizing problem solving activities. At the same time it raises concerns relating to the ability of a group of individuals to function as a coherent entity with collective intelligence, rather than a collection of independent agents (Smith 1994). An important requirement for the achievement of a state of collective intelligence is a high rate of information transfer among individual group members.

Newell (1990) argues that the communication bandwidth among human beings is insufficient for the members of a group to share the same knowledge; - a condition that he considers to be a prerequisite for collective intelligence. This would appear to be a matter of definition and degree. Smith (1994) argues that individuals do not necessarily utilize all of the relevant knowledge that is available to them when making decisions. There does not appear to be an a priori reason why complete shared knowledge is required for a group to achieve collective intelligence. Also, if the principal objective is for the group to devise a solution to the problem, then the purpose of collective intelligence is to ensure the existence of an adequate level of coherence within the group. While Newell's premise cannot be disputed, from a practical point of view we may be more concerned with degrees of collective intelligence. The information transfer rate must be sufficiently rapid to ensure that differences in relevant knowledge remain relatively small.

This view of collective intelligence further supports the distinction that can be drawn between cooperation and

collaboration. Collaboration assumes a high level of coherence among individuals as the group pursues a common goal. Each individual member of the group has some knowledge of the global solution objectives. Cooperation, on the other hand, has less stringent requirements for intellectual coherence and shared knowledge. The individual members of the group cooperate by carrying out their individual tasks without necessarily having knowledge of all contributions to the project. In this sense, collaboration could be regarded as a more sophisticated form of cooperation and, in fact, most groups tend to display behaviours that range between cooperation and collaboration.

1.1 Application: Cooperative Problem Solving

The ability of distributed, cooperative computing systems to harness the computing power of several machines is not the only reason for their utilization. Certainly, in the case of computation intensive applications the need for parallelism can be accommodated by assigning computational components to different processors. While computation may be a signifcant component of a complex problem system it is rarely the dominant feature.

The interests of the CAD Research Center in distributed cooperative computing were driven by a different set of parameters. First, we consider the user(s) to be an essential component of a computer-based decision-support system. For reasons discussed in previous publications (Pohl et al. 1994, Pohl and Myers 1994), much can be gained through the interaction of the human decision makers with the various computer-based components. In a human-computer partnership there are continuous opportunities for input, intervention and redirection, thereby circumventing many of the problems that have retarded progress in the field of machine learning. The presence of the human element, however, also brings with it the expectation that several users will participate from different locations. Similarly, the information sources that are relevant to the problem situation are likely to be distributed over several computing resources.

Second, the need for parallelism is not due to computational load but the large number and importance of the interrelationships that exist in a complex problem environment. The fact that a decision in any one problem domain may impact several other domains suggests that agents should operate in parallel. In this respect the agents emulate some of the behaviour of individuals participating in a group meeting. Proposals should spontaneously precipitate evaluations and countersuggestions, within a dialog format. The parallelism is essential here not only for reasons of preformance, but to prevent the system from pursuing solution strategies that do not adequately consider certain viewpoints.

Third, the interactions of the user(s) with the system will create conditions that will require the system to respond with actions from several domains. These responses have to be prepared in

parallel for at least two reasons. First, they must be received in a timely manner to prevent the user(s) from embarking on a solution path that has a high probability of failure. Second, the consensus positions that become available after the initial responses are of even greater importance to the user(s), and must be received quickly.

Finally, the users should be able to work on distinctly different aspects of the problem at the same time. This presupposes the existence of multiple computer workstations and adequate computer-based resources (e.g., agents) to service disparate user needs concurrently.

Clearly, these considerations suggest that from the perspective of decision-support systems the communication facilities are an integral component of the application. Their principal purpose is to 'enable' the interactions among the agents (including the user(s)), rather than distribute the computational load.

1.2 Implementation: System Considerations

From a hardware point of view, a distributed computing system typically involves a set of computers that are connected together by means of a network and collectively solve a large (computations) or complex (relationships) problem. The computers may be a heterogeneous mix, providing the opportunity to assign processors with different functional capabilities to perform different tasks or process different types of data (Durfee 1988).

From a software point of view, a distributed problem solving system may be defined as a loosely coupled network of semi-autonomous agents that cooperatively participate in the solution of a complex problem. The agents typically operate on their domain knowledge of the problem, the suggestions of other agents that are relevant to their domain, and any data resources that they are able to access. In this context, users are also agents that differ from the computer-based agents in respect to the superior intellectual powers and external knowledge that they bring to bear on the problem solving system.

The advantages of distributed computing systems are mostly economical and functional in nature.

- -- A number of connected smaller computers are less expensive than one large computer.
- -- A group of computers is more reliable than a single computer, since the failure of one does not necessarily jeopardize the operation of the entire system.
- -- Parallel execution allows sub-problems to be operated on concurrently, and the results to be

shared among components. This normally results in faster solution of the overall problem. In addition, it might be argued that parallel execution can simplify scheduling. Some cycles on separate processors may be wasted without concern for the delay of critical processes.

-- The results of concurrent operations can be shared among components of the system (including the users) in a timely fashion so that the actions of one contribute to the actions of the others during and not after individual actions have been completed.

-- A distributed system provides accessibility to users and data resources that is both convenient and practical. In other words, the distributed human and data resources are integrated operational and functional components of the system.

Difficulties posed by distributed systems are mostly related to matters of communication and coordination. Components operate asynchronously and concurrently subject to some inherent inter-node communication limitations. These limitations are related to bandwidth restrictions of the communication medium and the computational overhead of packaging and transmitting the information that is sent and received by agents (Durfee 1988).

Coordination problems are related to the intrinsic characteristics of cooperative problem solving, namely: limited communication; the possibility that individual agents may not have sufficient information to solve their sub-problems adequately; sufficiently reliable truth maintenance to ensure that all agents are accessing a common view of the current state of the problem that accurately reflects their collective contributions; deficiencies in the global view of individual agents; and, all of the difficulties that pervade group activities, such as the detection and control of contention, deadlock, fragmentation, interference, repetition, and non-convergence.

Control and coordination of the interacting agents is by far the most complex and interesting aspect of distributed cooperative systems. Not only must each individual agent know how it should most effectively apply its capabilities and local resources, but the agents as a group must also coordinate their activities to maximize the use of network resources. Interaction among agents can be highly productive or it can be a hindrance to the formation of any kind of useful collaboration. The more complex the interdependencies between problem domains the greater the potential for cooperation and, conversely, for obstruction.

Formal theories and protocols for the coordination of cooperation have been a subject of research for some time, with a significant

increase in activities from the 1980s onward (Chaib-Draa et al. 1992). The principal impetus of this increased activity has been the proliferation of networked computers in educational, commercial and government environments and the need and willingness of commercial and government organizations to address complex problem situations in the management, planning and design spheres.

In general, these theories have focussed on the application of predicate logic to the understandings, objectives, and actions of agents. For example: formal structures for reasoning that allow agents to formulate individual plans that take into account the views of other agents and can therefore be effectively combined into multi-agent plans (Georgeff 1983 and 1984, Lansky 1985); common knowledge and the ability and inability of agents to converge on common views (Halpern and Moses 1984), and identification of situations that may or may not lead to convergence (Rosenschein and Genesereth 1987); and, formal models of cooperative agent activity using game theory (Rosenschein and Genesereth 1985). While these theoretical studies tend to be useful for establishing boundaries for what is and is not possible, and for identifying research directions, they provide less help for the design and implementation of actual distributed cooperative problem solving systems.

From a more pragmatic perspective Durfee (1988) identifies four general goals for the design of such systems. These goals are representative of a view of cooperative computing that is widely shared. However, it is not the only view and therefore the goals listed below do not constitute a set of requirements. First, to facilitate the completion of problem solution tasks through concurrent activities. Typically, this entails decomposition, development of sub-solutions in parallel, and the application of coordination strategies to minimize the time agents have to wait before they receive contributing results from other agents. In addition, this first goal implies the need for a strong emphasis on local problem solving capabilities. The ability of an individual agent to develop sub-solutions in a semi-autonomous manner, with regard to but not controlled by the concerns of other agents, encourages solution alternatives and communication selectivity.

Second, to maximize the potential for task accomplishment through the sharing of resources (i.e., information, skills, etc.). This goal suggests the need for agents to share predictive information, to exchange tasks, and to assist each other in the testing and evaluation of results. The opportunity for sub-solution verification by agents from several different domains, each with its own knowledge, rules and prototypes, is of particular significance.

Third, to increase the task completion success rate through redundancy and experimentation with alternative solution procedures. Important tasks might be assigned to multiple agents to maintain a high level of reliability even though a local node may fail. A particular sub-problem might be assigned to several agents

that will attempt to solve the same task using different procedures, knowledge, and delegation strategies.

Fourth, to minimize the potential for inter-agent interference through the anticipation and avoidance of counterproductive interactions. Of particular importance in this regard are the maintenance of a quasi real-time communication environment, the ability of agents to recognize unnecessary and conflicting tasks, and selectivity in respect to the messages that are sent to other agents or broadcast to the system as a whole.

Conceptually, the protocols that are useful for coordinating a network of problem solving nodes in a distributed computing environment can be drawn directly from human interactions. What is immensely difficult is to effectively implement these protocols within the current constraints of computing systems. Foremost among these constraints is the limited amount of knowledge that can be made available in such systems. A secondary constraint is related to the rather primitive nature of the capabilities that agents can be endowed with. Typical types of coordination protocols that have been tested and are being progressively refined in actual distributed computing systems include:

-- Contract-Net Protocol: Agents decompose problems into sub-problems and engage other agents to cooperatively solve the sub-problems, utilizing a process of negotiation (Smith 1980, Davis and Smith 1983).

-- Multi-Stage Negotiation: Extension of the contract-net protocol to permit iterative negotiation (Conrey et al. 1988, Larsi et al. 1990).

-- Cognitive Model: Development of a computational model of interaction that produces compromise agreements among agents (Sycara 1989, Durfee and Montgomery 1990, Kraus and Wilkenfeld 1990).

Generally speaking the underlying strategy in all of these protocols is to decompose the problem into sub-problems that are domain specific, distribute the sub-problems to several agents, and provide a framework for these cooperating agents to collectively assemble the sub-problem solutions into overall solutions. The procedures used to support these decomposition, iteration and restructuring functions are typically simplified versions of those used in human interactions (e.g., information acquisition, evaluation, testing, bidding, negotiation, and planning).

1.2.1 Heterogeneous Network Computing

Distributed computing systems that operate in a network environment can present several kinds of heterogeneity challenges to the developers. First, the computer mix can include a wide range of architectures such as high-performance workstations, personal

computers, mainframes, and even shared memory multi-processor machines. Each architecture type has its own esoteric characteristics and preferred programming methods. In addition, there may be system software and, in particular, compiler differences that may require application modules for parallel tasks to be customized before they can be compiled on each machine.

Second, the data formats used by different computers may be incompatible. The encoding and decoding of information sent from one computer to another, so that it is readable to each node of the distributed system, becomes the responsibility of the communication facility. Third, there is a need to balance the computational speed and load on individual machines with the network load and the response requirements of the cooperative application. While the inherent computational speed of a computer will vary with architecture type, manufacturer and model, the actual computational speed is a direct function of the current workload. By their very nature networked computers are normally not dedicated to a single application, but serve the needs of multiple users. At least some of these users may be performing tasks that are unrelated to the distributed application.

The time it takes for information to be transmitted among computers will depend largely on the current network load. Much of this load may be generated by the external activities of users that are operating on network nodes. Even though these nodes may be machines that are not involved in the given distributed application, the increased network load will impact all users on the network. Particularly in the kinds of distributed cooperative systems described in this Report, where agents are designed to respond opportunistically to changes in the state of the problem, heavy network load can greatly impact the operational efficiency and behavior of the application.

1.3 Parallel Computing Communication Models

Parallel computing has been implemented in several ways, to meet the demands for higher performance, lower cost and sustained productivity. Massively Parallel Processors (MPPs) typically incorporate several hundred execution units, connected to gigabytes of memory, in a single computer. In distributed systems, on the other hand, two or more single computers are connected through a communication network. In addition, combinations of these two hardware configurations have been implemented to meet specific performance requirements. For example, several MPPs have been combined in a distributed computing network to achieve enormous computational capabilities.

Central to the notion of parallel processing is the ability to exchange data between cooperating tasks. A number of parallel computing communication models have been implemented and tested in different hardware environments and under varying conditions over the past two decades. Some were designed for a specific hardware

configuration, such as MPPs, and others have been implemented over a wider range of parallel hardware configurations. The situation is complicated by the fact that the different approaches can be categorized relative to several parameters: the physical disposition of memory (e.g., shared or distributed); the proportion of the total communication that is hardware-based or software-based; the precise nature of the unit of execution; and so on. The following sections provide a brief description of the major alternatives and their typical spheres of application.

1.3.1 Vector Processors

Vector machines began the age of supercomputers. They provide for the parallel processing of an array of similar data in a single operation. Even with the extension of this concept to include the operation of entire programs on several data structures, the fundamental notion remains unchanged. The parallelism is limited to the data, while the program that operates on the data is essentially sequential in nature. Accordingly, this computational model is not applicable to distributed computing.

1.3.2 Shared Memory

In the shared memory model every processor has access to a single shared address space. Access conflicts are typically controlled by some form of 'locking' mechanism that may or may not be transparently managed by the hardware and/or programming language. Variations of this model may include both shared memory that is accessible by all processors and local memory that is dedicated to a single process. As the name implies this computational model assumes the existance of some form of common memory space.

1.3.3 Message-Passing

The message-passing model provides facilities for sending and receiving data from the local memory space of one process to that of another. Without the requirement for shared memory this model can be applied to both single MPP machines and distributed networks. Its limitations are related to network transmission speeds and the processing overhead associated with the operations that have to be performed by both the sending and receiving processes.

1.3.4 Remote Memory Operations

This computational model has features of both the shared memory and message-passing models. One process can access the memory of another process without requiring the other process to undertake any action. However, the remote memory access must be explicit and is therefore more restrictive than local memory access. The notion

of remote memory access has been extended to include 'active messages' that automatically trigger the execution of an action in the memory of the remote process (Eicken et al. 1992).

1.3.5 Threads

All of the computational models mentioned above deal with processes as the entities that communicate with each other. In this context a process typically comprises an address space and a current state consisting of one program counter, register values and a call stack. Such an arrangement can be referred to as a single-threaded process.

Whereas early versions of the shared memory model provided each process with its explicitly defined separate address space, the common version today shares all of the memory. This allows several programs (i.e., processes) to be executed on a single processor in a timesharing mode. Each set of program counter, register values and stack, is referred to as a 'thread'. In this respect the principal difference between a thread and a process is the fact that a thread does not have its own address space.

The objective of the thread model is to provide for virtual parallel program execution within a single process, taking advantage of rapid control switching from one thread to another with minimum memory management overhead. While still early in its development, the thread model promises several potential advantages. Foremost among these is the natural manner in which non-blocking communication and shared memory operations can be implemented (Gropp et al. 1994).

1.3.6 Hybrid Models

Many combinations of the above models have been studied and several have been successfully implemented. For example, workstation and personal computer (PC) vendors are increasingly offering shared memory multi-processor machines that are readily integrated into networked distributed message-passing environments.

1.4 The Message-Passing Model

All of the inter-process communication systems used to date by the CAD Research Center in its various implementations of the ICDM (Intelligent Cooperative Decision Making) framework (Myers and Pohl 1994) have been broadly based on the message-passing model of parallel computing. Of course, strictly speaking, the blackboard architecture combines features of shared memory with the message-passing model. For example, early implementations of the ICDM framework did not permit direct agent to agent communication but required all messages to be channeled through the blackboard. However, even under those circumstances message-passing remains the

basic computational model. It was chosen not for any intrinsic superiority, but because of its natural compatibility with our notion of opportunistically cooperating agents in desision-support systems and its ease of implementation in distributed computing environments.

Nevertheless, the message-passing model has been widely adopted as the communication model of choice for parallel processing applications (Geist et al. 1990, Beguelin et al. 1991, Bomans et al. 1990, Butler and Lusk 1992, Gropp and Smith 1993, Skjellum et al. 1994, Harrison 1991). The reasons are related to both flexibility of application and performance (Gropp et al. 1994).

1.4.1 Flexibility of Application

The message-passing model is highly suitable for parallel computing environments where separate processors are connected by means of a communication network. These environments include both parallel supercomputers and workstation networks.

In addition, object-based applications utilizing the C++ language or object extensions of rule-based languages can incorporate a message-passing paradigm. For example, more recent implementations of the CAD Research Center's ICDM framework are object-based with an underlying distributed blackboard architecture that includes external message-passing to C++ and CLIPS agents, as well as internal message-passing within a single CLIPS process comprising multiple agent modules.

In distributed cooperative applications, message-passing is intrinsically suited to emulating the communication activities that occur in human problem solving collaborations. At the same time message-passing provides a comprehensive model for expressing parallel algorithms, including self-scheduling and adaptive algorithms.

While the detection and correction of errors in parallel applications remains a serious obstacle, experience appears to support the contention that the debugging process is somewhat easier in the message-passing model than in the shared memory model. This argument is based on the explicit manner in which the message-passing model controls memory references. Since only one process has direct access to any memory location at any particular time it is often easier to detect 'memory overwrite' conditions.

1.4.2 Performance

The potential performance advantages of the message-passing model are a direct outcome of the intrinsic memory management control capabilities provided by this computational model. As computers become faster, memory operations and the management of cache gain in importance. The ability of the message-passing model to

associate specific data with any particular process provides the vehicle by which this computational model can potentially make full use of the compiler and cache management facilities of the computer. This is an advantage not only in distributed memory computers that typically provide more memory and cache than the largest single-processor machines, but also in shared memory machines where the increased programmer control of data location in the memory structure can lead to significant performance gains.

2. The Original ICADS Communication Facility

In the ICADS (Intelligent Computer-Assisted Design System) prototype for building design six or more rule-based expert systems (the precise number depending on the current system configuration) execute in parallel as they interact through a data-blackboard during the design process. The objective of the system is to assist the building designer in creating a floor plan that satisfies some general architectural constraints and specific project requirements.

Since the ICADS prototype has been described in considerable detail in previous publications (Pohl et al. 1988, 1989 and 1994 (68-94)) and the focus of this Section is on the communication facility alone (Taylor 1990), it suffices here to mention only that as the user constructs a floor plan in the CAD-drawing environment the expert systems (i.e., agents) continuously and spontaneously monitor the evolving design solution, evaluating the impact of user decisions, suggesting solutions and reacting to counter-suggestions made by another agent acting in a conflict identification and resolution role. All of the agents send and receive information relating to the current state of the design solution opportunistically, as soon as it becomes available. Conflicts are resolved through discourse among the agents, although all communications are required to pass through the data-blackboard.

In the ICADS prototype truth maintenance is achieved by ensuring that information changes are communicated immediately to all concerned parties. Our experience with several ICDM (Myers and Pohl 1994) applications over the past decade has shown that if the information flow among the various components of a cooperative system is sufficiently fast, and spontaneous rather than scheduled, then incorrect and obsolete information is quickly corrected as new information enters the data-blackboard. Under these circumstances it is an important requirement for the communication system to be able to interrupt the current activities of an agent when new information has become available. At the same time, it is also important for the system to understand what kind of information may be of interest to a particular agent so that agents are not needlessly interrupted in their activities. This is achieved through a simple procedure at the beginning of any ICADS session, which requires each agent to register with the blackboard control agent an input template indicating the kind of information that it is interested to receive.

2.1 Message Handler

The four primary components of the ICADS blackboard control component are the Message Handler, the Attribute Loader, the Conflict Resolver, and the Geometry Interpreter. The Attribute Loader initializes a semantic network, representing the current state of the design, with knowledge about the type of design

project under consideration (Pohl et al. 1989 (89-91)), Renzetti 1995). The Conflict Resolver identifies conflicts and enters into a dialog with the agents in an attempt to arbitrate and resolve conflicts by priority or consensus (Pohl et al. 1989 (91-94)). The Geometry Interpreter maintains a high level object representation within the system, through both predefined rules and interaction with the user (Taylor and Pohl 1990). These components are not part of the communication system and will therefore not be discussed further in this Report.

The Message Handler acts as a central hub within the communication framework, by routing information between all system components. The communication framework supports multiple hierarchies of connections among both 'C' language (Kerninghan and Ritchie 1988) and rule-based processes written in CLIPS (NASA 1989). Each connection provides an independent two-way stream communication path between processes using UNIX sockets (Stevens 1990).

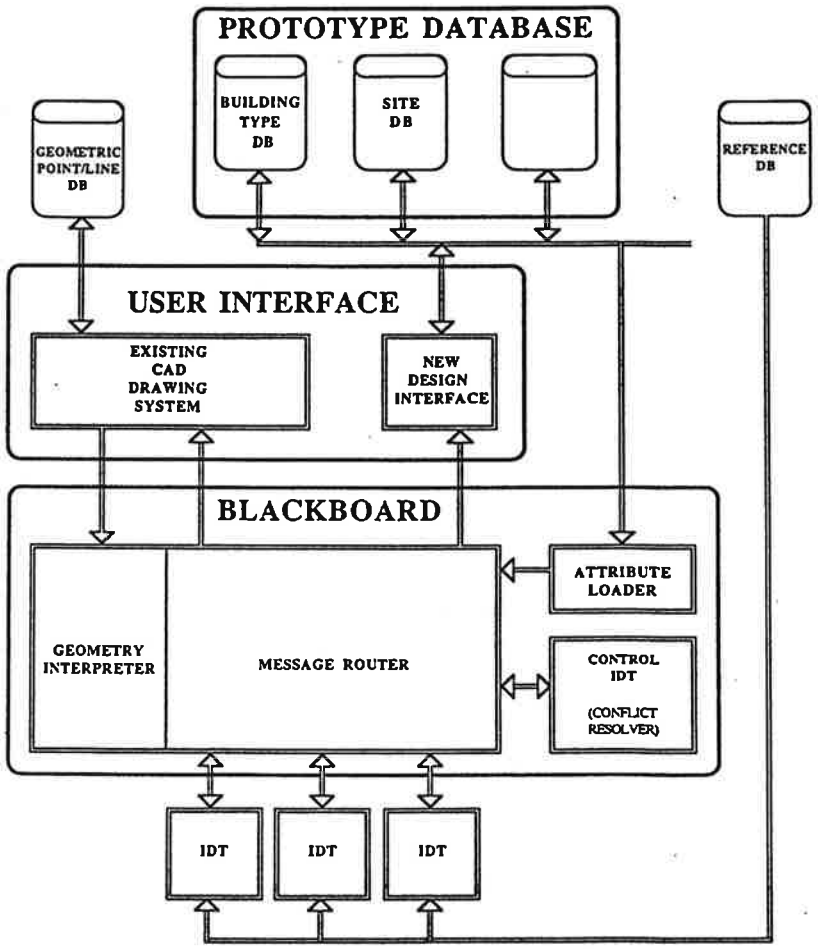


Figure 2.1: ICADS System Diagram

The network of connections within the ICADS prototype is shown in Figure 2.1. From the point of view of the blackboard's Message

Handler, the Conflict Resolver consists of a single connected component. However, to increase performance, the rule-set of the Conflict Resolver was divided into several independent rule-sets and distributed as separate processes across the network. The User Interface was also divided into two processes to take advantage of the organizational power of the RETE network (Forgy 1982) in CLIPS and the graphical display capabilities of X-Window (Jones 1988).

The part of the blackboard called the Message Handler is a CLIPS expert system with additional functions for message passing. The Message Handler has two primary functions. First, it initializes the system by starting each agent (referred to as IDT (Intelligent Design Tool) in Figure 2.1). Second, it distributes modified values to agents that request them. The Message Handler initializes the system in two phases. During the first phase it establishes a connection with each agent to allow message passing, and receives the input requests specifying the blackboard values the agent would like to receive so that it can produce its results. During the second phase, the Message Handler builds a hash table and transmits it to each agent to reduce the size of all future messages. An important prerequisite in this framework is that all system components use the same naming convention. Without a consistent naming convention, too much time would be spent converting between different representations. This common naming scheme is provided by a frame-based representation developed as part of the ICADS project (Assal and Myers 1990).

2.1.1 Representation

The particular frame based representation used in ICADS is implemented as a set of CLIPS facts. A frame is a collection of information about a class or object. The information is represented in CLIPS with a frame header fact and any number of slot facts. Slots can define a particular value of the class or identify a has-a relation to another class. A frame header is a fact of the form:

```
(FRAME <class> <instance>)

where: FRAME is a keyword
       <class> is the name of the class of this frame
       <instance> is the frame identification number
```

The FRAME header is useful in performing operations on the entire frame, such as deleting the frame, but is not needed to access the slots within the frame. A value slot is a fact of the form:

```
(VALUE <class> <attribute> <instance> <value>)

where: VALUE is a keyword
       <class> is the name of the class of this frame
       <instance> is the frame identification number
       <attribute> is the slot name or attribute
```

<value> is the actual value of the slot

The <value> field may be one or more values, depending on the nature of the slot. For example, a slot for the coordinate of a point would have two values, whereas a slot for the length of a wall would have only one value. A relation slot is a fact of the form:

(RELATION <class1> <class2> <instance1> <instance2>)

where: RELATION is a keyword
<class1> and <class2> are the names of classes

<instance1> and <instance2> are frame identification numbers of <class1> and <class2>, respectively

An example of an architectural object is the room or space object. Shown below is an instance of the class space with an identification number of 15, a name of LOBBY, a center coordinate of (128, 384), a perimeter of 108 feet, and four walls:

```
(FRAME space 15)
(VALUE space name 15 LOBBY)
(VALUE space center 15 128 384)
(VALUE space perimeter 15 108)
(RELATION space wall 15 1)
(RELATION space wall 15 2)
(RELATION space wall 15 3)
(RELATION space wall 15 4)
```

Changes to existing frames are made by inserting an action as the first field of the slot. Slots can be added, deleted, and modified using the keywords ADD, DELETE, and MODIFY. The ADD action asserts the slot, the DELETE action retracts the slot, and the MODIFY action retracts the existing slot and asserts a new slot. For example, if the above instance of a space class exists and (MODIFY VALUE space area 5 216) is asserted, then the following actions occur:

```
(VALUE space area 5 108)          ...is retracted
(VALUE space area 5 216)          ...is asserted
(MODIFY VALUE space area 5 216)   ...is retracted
```

When the DELETE action is asserted with the frame header, the entire frame (i.e., all slots and the header) is retracted.

2.1.2 External Functions

The external functions added to CLIPS to implement message passing are divided into two categories; namely, initialization and transmission. Messages are composed of any number of slots (i.e., CLIPS facts), and are received explicitly with an external function that asserts the slots in the message. Messages are built with

commands that were added to the standard CLIPS command set and have the same syntax as the CLIPS assert command. The following functions are used during initialization:

(new_server <name of process>)

Called by the Message Handler and the agents to create a server to allow future connection. Returns zero if no error has occurred.

(connect_bb [<name of message handler>])

Called by an agent to establish a two-way connection between itself and the Message Handler. Returns the identification number of the agent.

(accept_idt)

Called by the Message Handler to establish a two-way connection between the itself and an agent. Returns the identification number of the agent.

(unaccept_idt <agent id number>)

Called by the Message Handler to terminate the connection between itself and the agent specified. Returns zero if no error has occurred.

(insert_hstring <field1> <field2> ...)

Called by the Message Handler and agents to add a string composed of the concatenated fields to the hash table. Returns zero if no error has occurred.

Similarly, the functions used during the transmission of facts are briefly described below:

(receive_message [<agent id number>])

Called by the Message Handler and the agents to receive a message in first-in-first-out order and to assert the facts in the message. Receives a message from only the Message Handler, if zero is supplied as the agent identification number. Receives a message only from the agent specified, if the agent's identification number is supplied. Returns zero if no error has occurred.

(bb_assert (<fact 1>) [(<fact 2>) ...])

Called by the agents to add facts to the message buffer. Uses the same syntax as the CLIPS assert command. Returns zero if no error has occurred.

(bb_end_message)

Called by the agents to send the message buffer that was built with the 'bb_assert' command to the Message Handler. Returns zero if no error has occurred.

(idt_assert <agent id number> (<fact 1>) [(<fact 2>) ...])

Called by the Message Handler to add facts to the message buffer of the specified agent. Separate message buffers are maintained to allow messages for different agents to be built simultaneously. Returns zero if no error has occurred.

(idt_end_message <agent id number>)

Called by the Message Handler to send the message buffer that was built with the 'idt_assert' command to the specified agent. Returns zero if no error has occurred.

2.1.3 Initialization

The Message Handler has two phases of initialization. In the first phase, it starts each agent, establishes a connection to allow message passing, and receives input requests specifying the slots the agent requires as input. Each agent sends its input requests as its first message in the form of input value slots in an agent frame. The following example illustrates the actions performed by the Message Handler and the Sound and Light agents during the first phase of the initialization process.

```
MESSAGE HANDLER:
      (new_server "mhandler")
      (system sound.start)
      (receive_message (accept_idt))
      (system light.start)
      (receive_message (accept_idt))

SOUND AGENT:
      (new_server "sound")
      (bind ?no (connect_bb mhandler))
      (bb_assert
        (ADD FRAME idt ?no)
        (ADD VALUE idt input ?no FRAME space)
        (ADD VALUE idt input ?no FRAME space name)
        (ADD VALUE idt input ?no FRAME space area))
      (bb_end_message)

LIGHT AGENT:
      (new_server "light")
      (bind ?no (connect_bb mhandler))
      (bb_assert
        (ADD FRAME idt ?no)
        (ADD VALUE idt input ?no FRAME wall)
        (ADD VALUE idt input ?no VALUE wall length)
        (ADD VALUE idt input ?no RELATION wall window))
      (bb_end_message)
```

As shown above, an optional argument is supplied to 'receive_message' to specify that the next message should be

received only from the most recently started agent. This prevents messages sent by previously started agents from being mistakenly received and interpreted as the input requests for the most recently started agent.

In the second phase of initialization, the Message Handler builds a hash table to decrease the percentage of time spent transmitting messages by reducing the amount of information sent across the network. This technique was found to reduce message sizes by a factor of four to five. The Message Handler builds the hash table from the input requests of the agents. The keyword and class name fields of the input request slots are concatenated into a string and entered into a hash table. Then, when an instance of that slot is added to the message buffer with 'bb_assert' or 'idt_assert', the string of consecutive words starting with the second field is converted to a hash code, transmitted across the network as an integer, and then converted back to the original string of words upon receipt. If the string cannot be found in the hash table, each field is transmitted as a sequence of separate words. To insure that the hash code is correctly converted back to the original fields, the Message Handler and all agents must have identical hash tables. Thus, even though an agent may never receive a particular slot, the slot name is still contained in the hash table of that agent.

Using the example given previously for the first phase, the following strings would be entered into the hash table of the Message Handler, the Sound agent, and the Light agent:

```
(insert_hstring FRAME space)
(insert_hstring VALUE space name)
(insert_hstring VALUE space area)
(insert_hstring FRAME wall)
(insert_hstring VALUE wall length)
(insert_hstring RELATION wall window)
```

When the slot shown below is added to the message buffer, the second, third, and fourth fields (i.e., VALUE space name) are converted to a single integer hash code, sent across the network, and converted back to the original three fields upon receipt of the message.

```
(bb_assert (MODIFY VALUE space name 5 RECEPTION))
```

2.1.4 Distribution

After initialization, the basic loop of the Message Handler receives the next available message, distributes the slots of the message to the agents that requested them, and then retracts the slots. The following rules accomplish this for VALUE slots:

```
(defrule receive-message
  (declare (salience 40))
```

```
  ?f <- (RECEIVE)
=>
  (retract ?f)
  (receive_message)
)

(defrule build-message
  (declare (salience 30))
  (VALUE idt input ?no VALUE ?class ?attribute)
  (?action VALUE ?class ?attribute ?id $?value)
=>
  (idt_assert ?no (?action VALUE ?class ?attribute ?id $?valu
  (assert (SEND FRAME idt ?no))
)

(defrule send-message
  (declare (salience 20))
  ?f <- (SEND FRAME idt ?no)
=>
  (retract ?f)
  (idt_end_message ?no)
)

(defrule loop-rule
  (declare (salience 10))
  (not (RECEIVE))
=>
  (assert (RECEIVE))
)
```

Similar rules send the FRAME header and RELATION slots. Assertion of '(DELETE FRAME idt <agent id number>)' causes the Message Handler to retract the frame and terminate the connection of the specified agent. This fact must be asserted for an agent to exit prior to receipt of '(KILL)' without causing an error. Assertion of '(KILL)' causes the Message Handler to distribute this fact to all of the connected agents and then exit. The agents exit upon receipt of this fact.

2.2 Communication Architecture

As shown in Figure 2.2, there are three levels of 'C' language modules below the agent message-passing adjunct in the communication architecture. At the lowest level in the hierarchy the MESSAGE module implements transmission of information between distributed processes using UNIX sockets. This module takes care of mapping the logical name supplied by a process into a network address, creating and binding the socket to this address, establishing multiple connections to a single socket, and receiving facts from distributed processes in first-in-first-out order. The next level is represented by the FACTIO module which implements reading and writing of the elements as CLIPS facts. This module hides the representation and means of transmission of the fact.

The third level in the hierarchy depends on the language in which the agent is written. CLIPS knowledge bases use KBIO, while 'C' language programs (i.e., the CAD system and the User Interface) use BBIO.

Either of these modules establishes a two-way connection between the Message Handler and an agent, and is also responsible for hashing and unhashing the static fields of frame slots. The KBIO module allows facts to be transmitted using the same syntax as the CLIPS assert command. The BBIO module allows facts in the frame format to be transmitted with a single 'C' language function call.

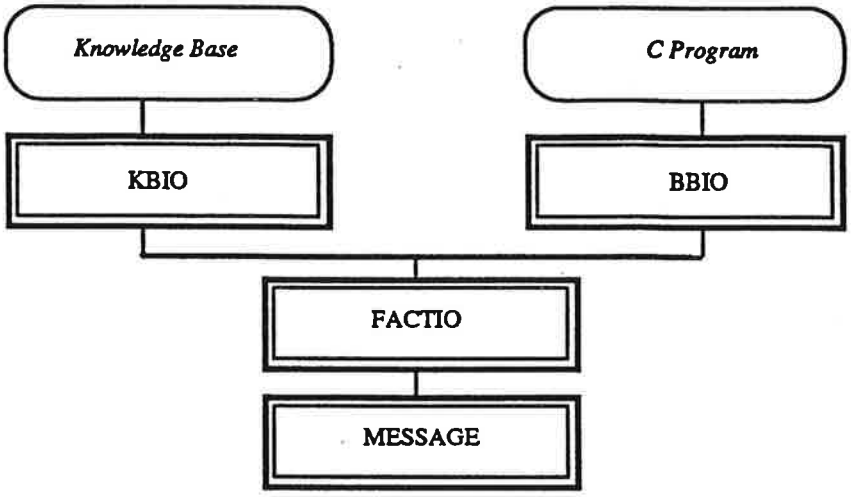


Figure 2.2: Communication Architecture Hierarchy

2.2.1 Message Module

The Message module (i.e., MESSAGE) is located at the lowest level of the communication hierarchy. It implements transmission of information between processes running on different computers that are in the same network using the UNIX socket library and various UNIX system calls. A socket, much like the UNIX pipe, is a stream communication pathway between processes. The advantage a socket has over a pipe is that it allows communication between processes running on different computers. Once a connection is established, a process uses the write UNIX system call to send data, while the other process uses the read UNIX system call to receive the data.

MESSAGE is based on a server-client model, in which the server can accept connections and receive data from any number of clients (Figure 2.3). The three logical divisions of this module are the creation of a server, the connection of a client to a server, and the transmission of buffered messages between them.

A server socket is created with the 'new_server' function. A text

string representing the logical name of the server is passed in as an argument. The 'new_server' function maps this logical name into a network address based on its entry in a configuration file. The machine and key entries of the logical name are combined to form the network address. Once the network address has been determined, the socket is created, bound to the network address, and enabled to accept connections. The return value of the 'new_server' function is a UNIX file descriptor. However, this file descriptor cannot be used for reading and writing. Instead, it is used to accept connections from clients as explained below.

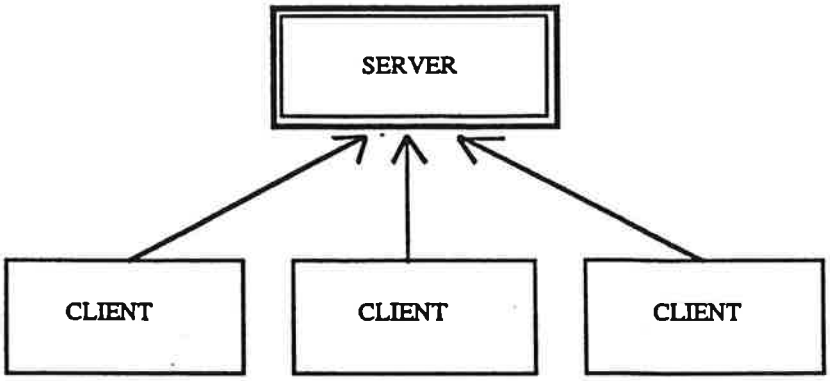


Figure 2.3: Server-Client Model Used in the Message Module

A connection is established by the server process performing an 'accept_client' call and the client process performing a 'new_client' call. The server must have previously performed a 'new_server' call for either of these functions to be successful. The 'accept_client' call requires the file descriptor returned by 'new_server' as its only argument. The 'new_client' call requires a text string representing the logical name of the server to connect to as its only argument. This logical name is mapped by the 'new_client' call into the identical network address as the server using the same technique as the 'new_server' call. A file descriptor that can be used in read and write system calls is returned by both the 'accept_client' and 'new_client' calls.

Both the server and the client block (i.e., pause) until a connection is established. This connection must be made at the same point in time, so that an accurate time offset between the processes can be measured. This time offset is subtracted from the sending time of a message to enable consistent time stamping of messages. Thus, messages can be received in first-in-first-out order, even when the processes are distributed across unsynchronized machines.

Information is transmitted through a socket in a logical grouping of bytes, commonly referred to as a message. The client process builds the message with the 'write_msg' call, which has the same arguments as the write system call:

```
write_msg (<file descriptor returned by new_client>,
           <address of buffer containing information>,
           <number of bytes in buffer>);
```

Each call to 'write_msg' has the effect of adding to the message that will be sent. The Message module imposes no limit on the size of a message. When the message is complete, the client sends it with the 'send_msg' call, which requires the file descriptor as its only argument.

The server receives messages using the 'receive_msg' call. Two arguments are required. The first argument is the file descriptor that was returned from the 'new_server' call. The second argument is either the constant ANYMESSAGE, which causes messages to be received from any of the connected clients in first-in-first-out order, or the file descriptor of a client, which causes messages to be received only from the specified client in first-in-first-out order. In both cases, if no messages are available, the server blocks. The return value of 'receive_msg' is the file descriptor of the client that sent the message. The server process reads the message with the 'read_msg' call, which has the same arguments as the read system call:

```
read_msg (<file descriptor returned by receive_msg>,
          <address of buffer to contain information>,
          <number of bytes to place in buffer>);
```

The return value of 'read_msg' is the number of bytes actually placed into the buffer. Thus, if an attempt is made to read past the end of a message, a zero will be returned.

An additional function 'query_server' is available to check whether or not any messages are available to be received, without causing the server to block. The arguments are the same as the 'receive_msg' call, and the return value of 'query_server' is zero if no messages are available and non-zero if messages are available.

The implementation of the transmission functions automatically prevents the occurrence of a deadlock. A potential for deadlock exists because a client process will block after writing a system defined number of bytes. Thus, if two processes simultaneously write enough information to each other to reach this limit, then they will both be blocked forever. This deadlock is prevented by receiving and queuing (internally) all available messages before a message is actually transmitted with the 'send_msg' call. This deadlock prevention procedure is completely transparent to the user of the Message module.

If an error occurs in any of the functions implemented in the Message module, then the constant ERROR (defined as -1) is returned.

2.2.2 Fact Input/Output Module

The Fact Input/Output module (FACTIO) hides the representation and means of transmission of a CLIPS fact. A fact is written using one 'write_header' call and a 'write_element' call for each field. The 'write_header' call requires two arguments:

```
write_header (<file descriptor returned from new_client>,
              <number of fields in the fact>);
```

The 'write_element' call requires four arguments:

```
write_element (<file descriptor returned from new_client>,
               <address of buffer containing element>,
               <type of element>,
               <number of elements represented>);
```

The possible element types are WORD, NUMBER, STRING, and HVALUE. The number of elements represented is equal to one, except in the case of the HVALUE type, where it is equal to the number of fields represented by the hash code being transmitted. For example, to write the fact '(MODIFY VALUE wall length 5 24)', the following calls would be used:

```
write_header (fd, 6);
write_element (fd, MODIFY, WORD, 1);
write_element (fd, &hvalue, HVALUE, 3);
write_element (fd, &instance, NUMBER, 1);
write_element (fd, &value, NUMBER, 1);
```

where: 'fd' is the file descriptor returned by the new_client call
'hvalue' is the integer hash code for VALUE wall length
'instance' is a float equal to 5
'value' is a float equal to 24

Both 'write_header' and 'write_element' use 'write_msg' to add information to the message buffer. By convention, the message buffer is terminated with 'write_header(fd, SENTINAL);', where SENTINAL is defined in the 'include' file of this module.

A fact is read using one 'read_header' call, and the 'read_element' call for each element. The 'read_header' call returns the total number of fields in the fact, and requires one argument:

```
read_header (<file descriptor returned by receive_msg>);
```

The 'read_element' call returns the number of fields the element 'read' represents, and requires three arguments:

```
read_element (<file descriptor returned by receive_msg>,
              <address to put contents of element>,
              <address to put type of fact>);
```

The following calls would be used to read the fact in the above example:

```
read_header (fd);                       /* returns 6 */
read_element (fd, word, &type);         /* returns 1 */
read_element (fd, &hvalue, &type);      /* returns 3 */
read_element (fd, &instance, &type); /* returns 1 */
read_element (fd, &value, &type);       /* returns 1 */
```

where: 'word' is set to MODIFY
'type' is set to WORD
'hvalue' is set to integer hash code of VALUE wall length
'type' is set to HVALUE
'instance' is set to float value 5
'type' is set to NUMBER
'value' is set to float value 24
'type' is set to NUMBER.

Both 'read_header' and 'read_element' use 'read_msg' to add information to the message buffer. When 'read_header' returns the constant SENTINAL, then no facts remain in the message buffer.

2.2.3 Blackboard Input/Output Module

The Blackboard Input/Output module (BBIO) provides a 'C' language interface for communication between agents written in the 'C' language and the Message Handler resident in the blackboard. Functions were implemented to provide the same message passing ability for 'C' language agents that is available through the external functions in CLIPS. Functions are available for initialization and transmission, as follows:

```
connect_bb (<name of message handler>, <name of IDT>);
```

Establishes an independent two-way communication path between the agent and the Message Handler. Returns the identification number of the agent.

```
build_bbhtable ();
```

Receives message and builds hash table from hash strings in the message buffer.

```
signal_bbmessage (<function>);
```

Sets SIGAIO signal to be invoked on receipt of a message, and installs '<function>' to handle this signal.

```
query_bbmessage ();
```

Checks if messages from Message Handler are available, without blocking. Returns zero if no messages are available and non-zero if messages are available.

```
receive_bbmessage ();
```

Receives message in first-in-first-out FIFO order from the Message Handler.

```
read_bbheader ();
```

Reads fact header from message buffer and returns the number of fields in the fact to be read.

```
read_bbelement (<buffer>);
```

Reads a fact element from the message buffer and stores it in '<buffer>'. Returns the type of field read, either WORD, NUMBER, or STRING. Automatically unhashes and buffers fields represented by hash codes, and thus 'HVALUE' is never returned as the type of field.

```
read_bbmessage (<buffer>,<number of bytes>);
```

Reads '<number of bytes>' from message buffer, and stores it in '<buffer>'. Used to read non-fact information from the message buffer.

```
send_bbheader ();
```

Sends the message buffer to the Message Handler.

```
write_bbframe (<action>,<class>,<instance>);
```

Writes frame header fact into message buffer. If 'FRAME <class>' is found in hash table, then a hash code is sent for this portion of the slot. If NULL is used as the '<action>' argument, then the header is transmitted without an action field.

```
write_bbrelation (<action>,<class1>,<class2>,<instance1>,<instance2>);
```

Writes relation slot fact into message buffer. If 'RELATION <class1> <class2>' is found in hash table, then a hash code is sent for this portion of the slot. If NULL is used as the '<action>' argument, then the slot is transmitted without an action field.

```
write_bbintvalue (<action>,<class>,<attribute>,<instance>,<integer>);
write_bbdimvalue (<action>,<class>,<attribute>,<instance>,<float>);
write_bbcoordvalue (<action>,<class>,<attribute>,<instance>,<coordinate>
write_bbwordvalue (<action>,<class>,<attribute>,<instance>,<word>);

write_bbwordvalues (<action>,<class>,<attribute>,<instance>,<word1>,
                    [<word2>, ...] NULL);
```

Writes value slot fact into message buffer. If 'VALUE

<class> <attribute>' is found in hash table, then a hash code is sent for this portion of the slot. If NULL is used as the '<action>' argument, then the slot is transmitted without an action field.

write_bbheader (<number of fields in fact>);

Writes fact header into message buffer. Used for CLIPS facts that are not in frame format.

write_bbelement (<buffer containing field>,<type of field>);

Writes field into message buffer. Used for each field of CLIPS facts that are not in frame format.

write_bbmessage (<buffer>,<number of bytes>);

Writes '<number of bytes>' starting at '<buffer>' into message buffer. Used to write information (other than facts) into the message buffer.

3. GXI: Graphical X-Window Interface Builder

By 1989, with the completion of the first substantial application of the ICDM framework in the form of the ICADS prototype for engineering design, the CAD Research Center was faced with two related development problems. First, as the scope of projects became more ambitious the development tools became more complex requiring team members to possess increasingly higher levels of programming skill and experience. There was a distinct danger that the CAD Research Center would gradually lose its interdisciplinary character as members from disciplines other than computer science would find it increasingly difficult to contribute on development teams. Second, the CAD Research Center recognized the need for user interfaces to become more and more graphical in nature.

Following industry trends in the UNIX workstation environment project groups chose to utilize the interface building facilities offered by the X-Window graphics system. More specifically, ICDM application interfaces were developed using the X-Window Athena, and later the Motif, widget sets (ASP 1988, Young 1990). However, the complexities of the X-Window development environment presented formidable difficulties to the domain expert members of the various development teams, unnecessarily diverting their attention from the application objectives to esoteric implementation issues. It was found that this situation not only dampened the motivation level of individuals, but also substantially reduced the productivity level of teams.

At that time the CAD Research Center development environment utilized mostly the 'C' programming language (Williams 1988) and the CLIPS expert system shell (NASA 1989). Neither of these languages provide high level facilities for the creation and manipulation of graphic entities. In the case of CLIPS, absolutely no facilities of this kind existed in 1990. However, CLIPS did (and still does) provide methods for incorporating user defined functions into the language. It is this flexibility which allowed the CAD Research Center to enhance the basic CLIPS shell to encompass graphical interface creation and manipulation facilities (Pohl 1991).

As discussed earlier in this Report, internal communication within a distributed execution environment is a fundamental requirement for cooperative decision-support systems. In highly interactive systems, such as the various implementations of the ICDM framework, real-time event handling becomes an essential facility if the responsiveness (and the integrity) of the system is to be maintained. In this context, real-time event handling is defined as the ability of the target system to react to the occurrence of an event within a specified amount of time (Krakowiak 1989). Such a characteristic becomes even more crucial when predominantly message-based systems are considered. Since the ICDM framework incorporates both of these characteristics, it became imperative to

develop a high level tool that would shield the application programmers from the implementation complexities of a real-time, distributed, inter-process, message-passing environment.

3.1 Motivating Factors and Design Alternatives

The Graphical X-Window Interface Builder (GXI) was developed in response to several growing needs. One of these needs dealt with the manner in which quantitative data are expressed. In order for acquired data to fully convey its meaning, the data must be presented in such a fashion as to reveal not only their value but also the context in which they appear. On the surface this may seem to be commonsense. However, much of the data representation seen in the late 1980s existed in the form of tabularized numbers or simply 2-D point/line graphs.

The CAD Research Center recognized that with the increasing availability of more sophisticated graphics capabilities on computer workstations data representation should no longer suffer from these limitations. Clearly, with the aid of advanced graphics programming tools, data could now be expressed in the form of 3-D graphs and solid images. Representations that take advantage of the ability of the human cognitive system to understand even complex graphical images more easily than textual data.

However, along with these added graphics capabilities comes the problem of trying to work with what usually amounts to a set of fairly complex graphics tools. The use of these tools tends to be time consuming and requires a great deal of programming skill. Accordingly, within the context of the CAD Research Center development environment it became highly desirable to create a set of simplified graphics request function calls. Moreover, these requests should exist largely as logical objects (e.g., menus or polygons) that contain all of the corresponding attributes.

The second motivating need for the development of GXI was the lack of graphical capabilities in virtually all artificial intelligence (AI) language environments at that time. In particular, the CLIPS expert system development shell used extensively by the CAD Research Center was totally devoid of graphics facilities. Therefore, our development teams were unduly restricted in their ability to incorporate graphical user interfaces into ICDM applications. To solve this limitation, it became desirable to extend the CLIPS language to include the interface facilities offered in environments such as X-Window.

Together, these requirements set the stage for the development of GXI. The general objective was to provide the programmer with a simplified object-based view of various menuing systems in addition to a wide variety of graphics primitives. Specifically, the facilities offered by the GXI environment should allow robust graphical user interfaces to be incorporated in clients that are written in either the 'C' language or CLIPS. In addition, by

providing its clients with a wide array of graphics primitives in combination with graphics objects GXI should allow for high level data expression in the form of 3-D graphs and similar images.

3.1.1 Single Process Approach

Two main approaches were considered during the design of GXI. The first approach is based on the concept of providing a set of client graphics calls along with their implementation in a single physical process. This approach brings with it some distinct advantages. The first advantage is that it allows the client requester and the graphics server to exist as one cohesive process. This allows all graphics actions to be centralized on a single machine thus freeing the programmer from the complexities of a networked environment. The second advantage again deals with reducing the degree of complexity. A single process approach would allow the programmer to link to a library of graphics routines. Therefore, programs would view GXI simply as a library of robust graphics routines.

However, a single process approach also has various inherent deficiencies. As mentioned earlier, both the graphics requester and the graphics server reside in the same process as one cohesive pair. Therefore, when there are several graphics applications running in the system there can be no logical or physical connection between them. Each exists as an independent entity completely ignorant of the others. This must inevitably lead to duplication of code and sequential processing of graphics requests.

Another deficiency arises when the application system resides in a networked environment. The single process approach has no provisions for dealing with anything but a local domain environment. This is incompatible with the very nature of an ICDM application, as a distributed cooperative decision-support system.

Perhaps, the most serious disadvantage of the single process approach is related to the X-Window graphics system itself. X-Window provides a collection of graphics primitives for the application programmer. However, these primitives exist in basic form and require extensive programming knowledge of the X-Window graphics environment to provide application facilities of any real value. The purpose of GXI is to utilize these tools extensively. However, the graphics tools provided to the GXI client exist at a considerably more abstract and less complex level than those that GXI uses to realize the client's graphics request.

In an effort to clarify the various constraints set forth by the X-Window environment, a brief explanation of the method used by X-Window to accept, perform, and reply to application graphics requests is necessary. When the application makes a request of the X-Window server an event is placed on an output queue contained on the server side. This output queue is unique to each application client and can be readily accessed by its owner. Event structures contain all the information required by the X-Window server to

bring into existence the particular request or event. When the application wishes to execute an event, it simply removes this prepackaged event from the queue and dispatches it through a series of calls to the X-Window server. To simplify this process even further, the application has the option of entering into an event loop which monitors the application's output event queue. As soon as an event is loaded into the queue it is dispatched. At any given time if there are no more events left in the queue, the application may block (i.e., sleep until another event arrives). These events are produced by activities such as a client graphics request or mouse interaction by the user with an active menu button.

Adhering to the single cohesive unit approach means that the X-Window server must give up control to the application once it has fulfilled the client's request. Situations where control is moved in a sequential fashion throughout the system, leaving various high priority functional entities inactive for extended periods of time, will inevitably lead to several serious synchronization dilemmas.

The first of these dilemmas is a peculiarity of most large scale graphic tools systems. As mentioned earlier, the X-Window system is a complex system consisting of several processes which have the ability to communicate with each other across a network. Using the single process approach, control moves from the client application to the request server following the issue of a graphics request. With control placed on the server side the client is put to sleep while the server attempts to carry out the particular request. This is analogous to the situation where a program wishes to read a number from the keyboard. Once the program has issued a 'read' system call, control moves from the program to the operating system. The client program is simply blocked until the user enters a value and presses the 'return' key. Once the user has entered the value, control is returned to the program. However, the operating system as a whole is still able to capture other requests or events independent of the particular client application. This is due to the fact that the operating system itself consists of several processes. Therefore, in the case of an operating system, control actually resides in several places at the same time. This allows the operating system to be receptive to activities other than a specific client's request at any given point in time. The flow of control structure described above is typical of most client-server relationships.

Since X-Window is based on the same client-server model, it is not difficult to relate this concept to the problem at hand. It is by no means a trivial matter to remove control from the X-Window system. Any attempt to artificially break the event catching and dispatching loop of X-Window can cause serious synchronization problems between the client and the server. For example, it is common for a series of menu creation events to be queued and sequentially dispatched resulting in the image of a menu without any buttons. X-Window must have control to block itself and consequentially wake itself up again. Since this is typical of many large scale graphic tool systems, GXI must be compatible with this

environment. A single process approach however, does not readily promote such an environment.

The above description of the movement of control throughout the system illustrates the second dilemma encountered in the single process approach. Once the graphics server has carried out the particular client request, there is no other option but for the server to relinquish control back to the client. This means that the server has literally been put to sleep. In this state, the server is no longer receptive to any mouse activity or other input/output events. Until the client makes another request to the server, the server is completely ignorant of any interactive activity taking place with the user. Since the very essence of an interactive interface is its continuous receptiveness to input/output activities, the single process approach is not acceptable.

3.1.2 Client-Server Approach

The design approach which best satisfies the requirements set forth involves multiple processes (Figure 3.1). In this approach, there exists a true graphics requests server that resides in its own process. As with the previously described model, client applications make graphics requests of the server which in turn performs the necessary work and returns any results (1.e., a handle to the graphics object).

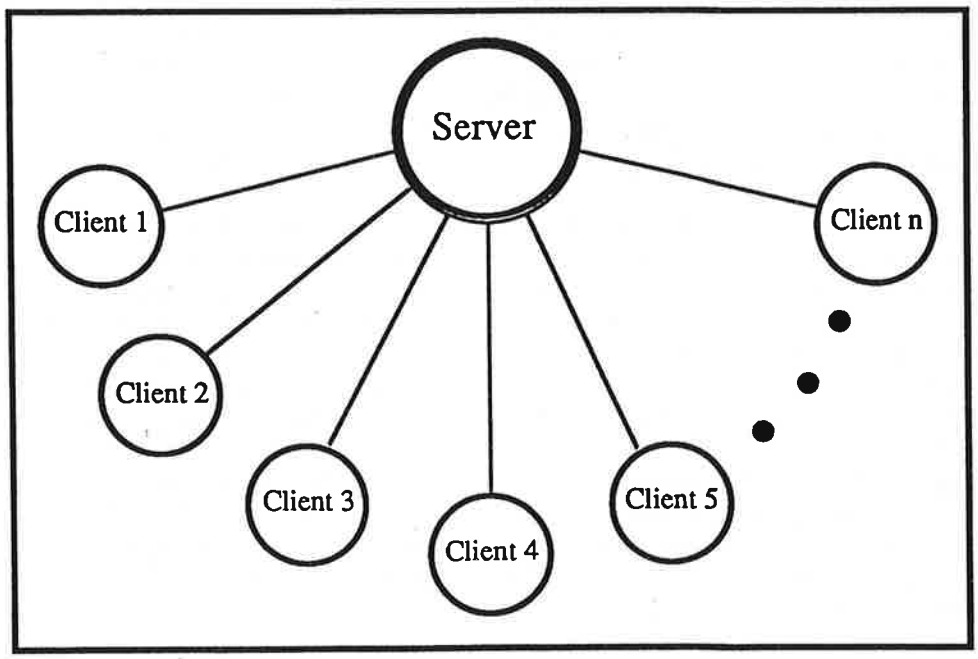


Figure 3.1: Client-Server Model

The clients making the requests have no concept of exactly how

their graphics requests are being carried out. Thus, the client application is removed from the complexities of internal graphics processing and representation. This approach proves to overcome all of the shortcommings of the single process approach. Further, this true client-server model provides a considerably more complete and robust design of the system allowing for inter-client communication and a fully networked environment.

The advantages of a client-server approach are basically threefold. The first advantage is related to the matter of control. As mentioned earlier, serious synchronization conflicts arise when control is forcably moved throughout the system. Using a client-server design, the server and all of its clients reside in their own processes. Therefore, as with the operating system example, control resides in several places in a concurrent fashion. These independent entities communicate with each other through the use of UNIX socket facilities, allowing the server the freedom of blocking itself and consequently waking itself up again. Therefore, the server can be receptive to any interactive activity precipitated by the user independent of the current state of its clients.

Second, the logical and physical connection which was lacking in the single server design is now present in an organized and complete fashion. There exists only one graphics server which performs all of the graphics work requested by each client. This is true no matter which client on which networked machine is actually making the request. The server can be thought of as an invisible workhorse running in background somewhere on the system. Each client simply requests a connection of this invisible entity at the beginning of the session. Since all requests are made to a centralized server, there now exists a logical and physical connection between each of the server's clients. Since all information is channeled through the server, graphics data can be easily passed from one client to another via the server.

The third advantage is related to the fact that ICDM applications are by their nature distributed, cooperative systems. In this respect they take advantage of the networked environment by distributing the workload among several workstations (Stevens 1990, Durfee 1988). GXI was designed to take full advantage of these benefits allowing each client to be positioned on any workstation connected to the network. Further, GXI does not require the client application to display its graphics on its own monitor. Since GXI is modeled in a true client-server fashion, it allows each client the freedom of selecting exactly which monitor it wishes its graphic work to be displayed on. In other words, from the point of view of the client the entire collection of networked workstations is simply one virtual machine.

In fairness, however, the client-server approach does have some disadvantages. In comparison to a single process system, the client-server system requires several added facilities for managing the networked environment. Some dynamic method must be provided to

allow the process-to-machine relationship to be altered as the network evolves, without requiring a recompilation of the system. This can be accomplished with the introduction of a configuration file that is read by the system at run-time. In addition to the various process-to-machine relationships this file may also provide the system with information pertaining to client display requirements. However, this is yet another parameter one must be concerned with in a networked client-server environment.

A second issue relative to the client-server approach deals with process synchronization. In a system containing multiple processes communicating with one another, there is an inherent potential for a deadlock. A deadlock occurs when one process is blocked waiting for a message from a certain process. However, that particular process itself may be blocked waiting for a message from the first process. The result is that both processes will remain blocked, or deadlocked, forever (Tanenbaum 1987). The same problem was discussed in Section 2.2.1 in a slightly different but related context. In a multiple process system a carefully designed communication protocol must be developed that will prevent such a situation.

In determining which approach would best fulfill the constraints placed on the GXI system, both the advantages and disadvantages of the single and multiple process strategies were taken into account. Considering the number of networking and functional entity synchronization advantages offered by a client-server model, the decision to develop GXI as a multiple process system was inevitable.

3.2 The GXI Architecture

The GXI server will except any number of client applications written in either the 'C' language or the CLIPS expert system shell language (Figure 3.2). In addition, these clients may execute on any machine connected to the network. The only requirement is that they list themselves in a configuration file that describes which network machine they wish their graphics work to appear on. This machine does not, however, need to be the same one that the client is actually resident on.

When a prospective client wishes to connect to the GXI server, it simply issues the '(XCInit)' or 'XC_Init()' commands for clients written in CLIPS or 'C', respectively. This command sends a connection request out across the network to the mother GXI server which is waiting in an accepting state for the next client request. Once the GXI server receives the connection request, it creates a child process in the exact image of itself known as a client request handler. The purpose of this handler is to carry out all further GXI requests issued from that particular client. The mother server then returns to its accepting state and waits for the next client connection request. The client request handler proceeds to make a formal connection to the X-Window system. Once this connection has been established, the handler proceeds to build a

color hashing table of all colors that the client intends to use throughout its session. This is done in an effort to increase performance and is discussed further in Section 3.4.2.

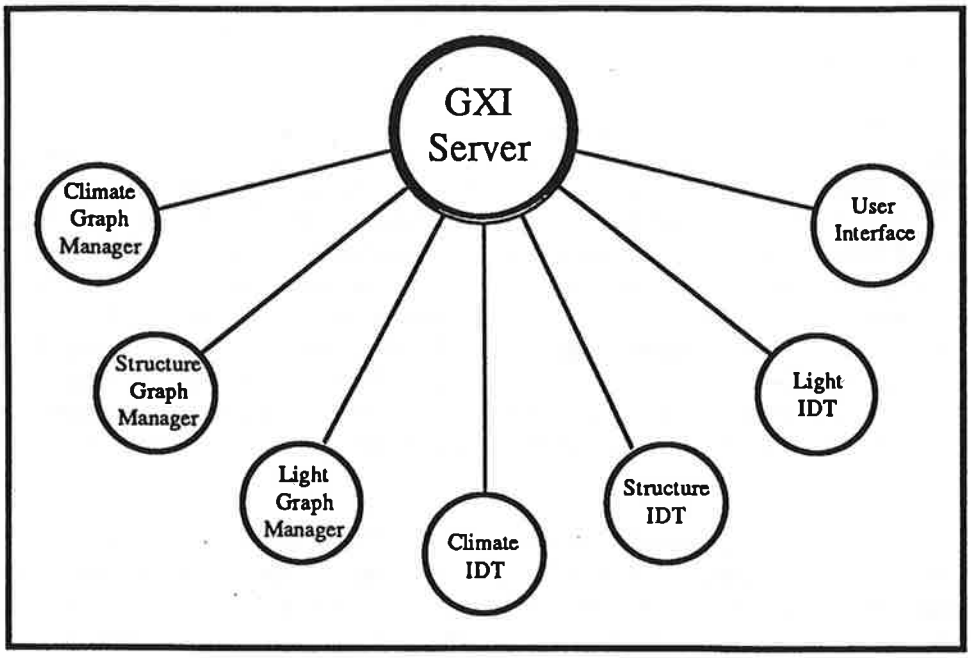


Figure 3.2: A GXI Application (ICADS)

Following the creation of the color table, the server proceeds to develop a working environment for the client. This environment includes an initial work sheet in the form of a graphics window. When the server has completed this initial environment setup, it returns the unique ID of the work sheet to the client via the GXI Communications module. When the client receives the results of its initial connection request it is then free to begin issuing its GXI commands to the GXI server. To terminate its connection to the server, the client simply issues a '(XCQuit)' or 'XCQuit()' command for CLIPS or 'C' programs, respectively.

3.3 Life of a Client Request

The GXI server, along with all of its clients, consists of several logical modules. In addition, both of these entities use various abstract data types (ADT) to assist them in managing their environments. To successfully execute a client graphics request, it is the client's task to formulate a message which it then sends to the GXI server. This message contains everything that the server needs to know in order to carry out the particular request. To gain a more comprehensive understanding of the way in which this message is formulated and subsequently used by the server, it is beneficial to trace the life cycle of a typical graphics request as it moves throughout the GXl environment.

3.3.1 The Client's View

The cycle begins with the client application issuing a graphics request to the server. This is accomplished by the client simply calling one of the predefined GXI function calls. Clients written in the 'C' language deal directly with the GXI Client Functions module. Whereas, clients written in the CLIPS expert system shell language deal with an intermediate module known as the GXI CLIPS Functions module (Figure 3.3).

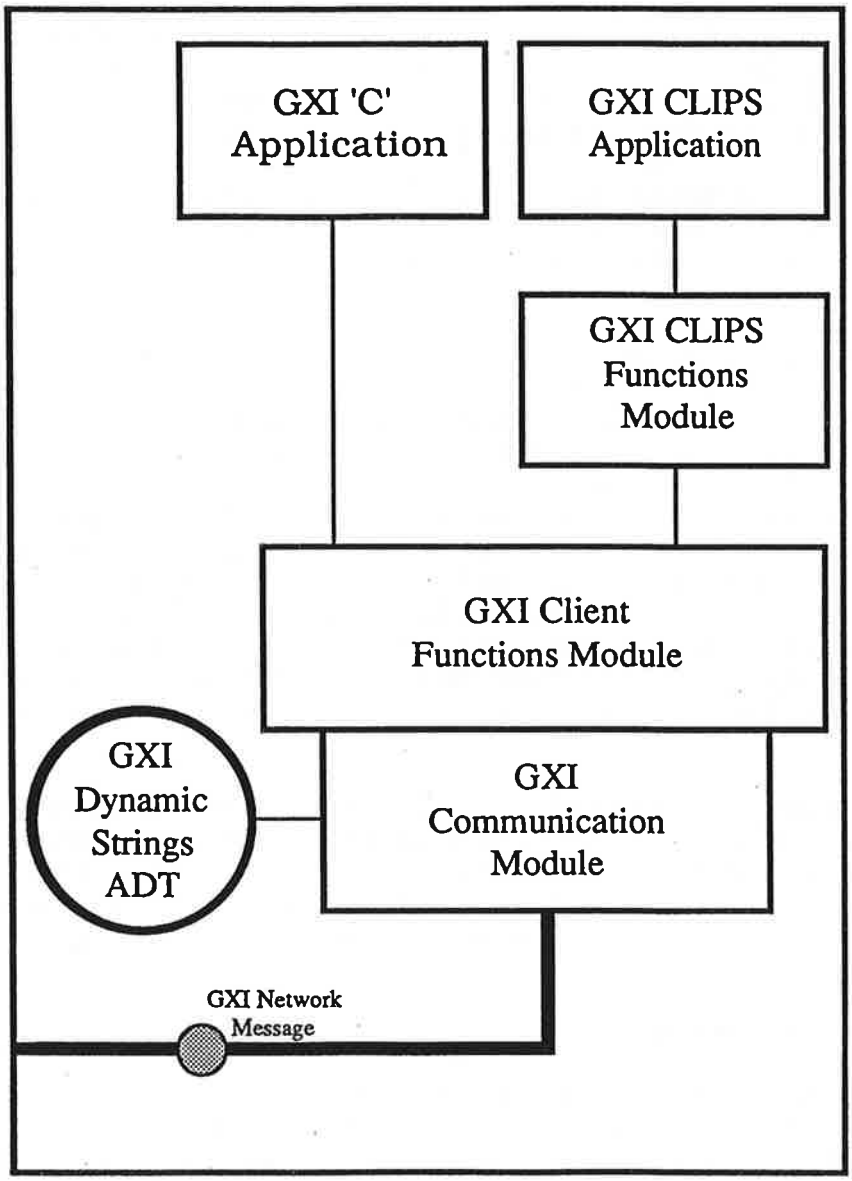


Figure 3.3: The GXI Client Architecture

GXI CLIPS Functions Module:

The purpose of this module is to extract out of the CLIPS environment all of the necessary arguments from the initial CLIPS function call that are needed to define the particular graphics action. This is accomplished using the external user functions facilities provided by the CLIPS expert system shell. A programmer wishing to add external functions written in 'C' to CLIPS is required to define the function names and corresponding return types in the 'usr_funcs' array facility provided by CLIPS. As part of the initialization performed by CLIPS this array of function names is read and entered into its list of possible application function calls. In the case of GXl these external functions are defined in the GXI CLIPS Functions module which is physically compiled into CLIPS. To form the entire GXI Client environment, the GXl Client Functions and GXI Communications modules are then linked in with this code to form GXIclips (NASA 1989).

Once all of the arguments have been successfully extracted from the CLIPS environment, a call is issued to the GXl Client Functions module, passing all of the newly acquired arguments.

GXI Client Functions Module:

The purpose of the GXI Client Functions module is to fill a message structure (see Appendix A) which will be sent to the GXl server. Since this structure is the only information that the GXI server will receive in regard to the particular graphics request, it must contain everything that the server needs to know to carry out the request. The message structure exists as a union of all possible GXI commands. In addition to this union the structure also contains several parameter fields that allow for dynamic data transfer, such as strings or coordinates, which define a region. This is described more fully under the section GXI Communications Module which follows below.

Finally, a unique GXI command code is placed in the tag field of the union to allow the command arguments to be selected from the correct union record after the message has been received by the server. Once the message structure has been successfully filled, it is sent to the GXI Communications module.

GXI Communications Module:

It is the purpose of this module to perform all of the communication between the client and the server. In addition, this module is responsible for packaging any dynamic string information into the proper format to allow it to be sent to the server. This is accomplished using the Strings ADT which concatenates any number of strings into one continuous stream of bytes. In order to preserve the individual strings, a string separator is placed

between each logical string. The size of this new stream is placed in a buffer size field located just outside the union structure of the message. At this point the initial message structure is sent out into the network on its way to the GXI server. Directly following this event, any prepackaged dynamic data are also sent to the server. Once all data have been sent, the client process assumes a waiting state for the server to respond with the results of its request.

3.3.2 The Server's View

GXI Communications Module:

After each client request has been carried out, the GXI server waits for the next client request. Once a request has been received, the server proceeds to read in any dynamic data indicated by the initial message structure. When all information concerning the particular client request has been successfully read, control moves to the GXI Server Main module (Figure 3.4).

GXI Server Main Module:

At this point the unique GXI Command Code that was placed into the initial message structure by the client process is used as an index into a function table. This table consists of pointers to all of the possible GXI command functions that reside in the GXI Server Functions module.

GXI Server Functions Module:

It is in this module that the client request is actually carried out. To complete its task, functions in this module have access to the GXI Widget Table and the GXI Color Table ADTs which are discussed in Sections 3.4.3 and 3.4.2, respectively.

Once the specific task has been completed a reply message structure is filled with the particular task results. This structure is then passed back to the GXI Server Main module and onto the GXI Communications module.

GXI Communications Module:

At this point the reply message structure is sent out, along with any dynamic data, across the network on its way to the client. Once the message has been successfully sent off, the GXI server simply returns to a waiting state for the next client request.

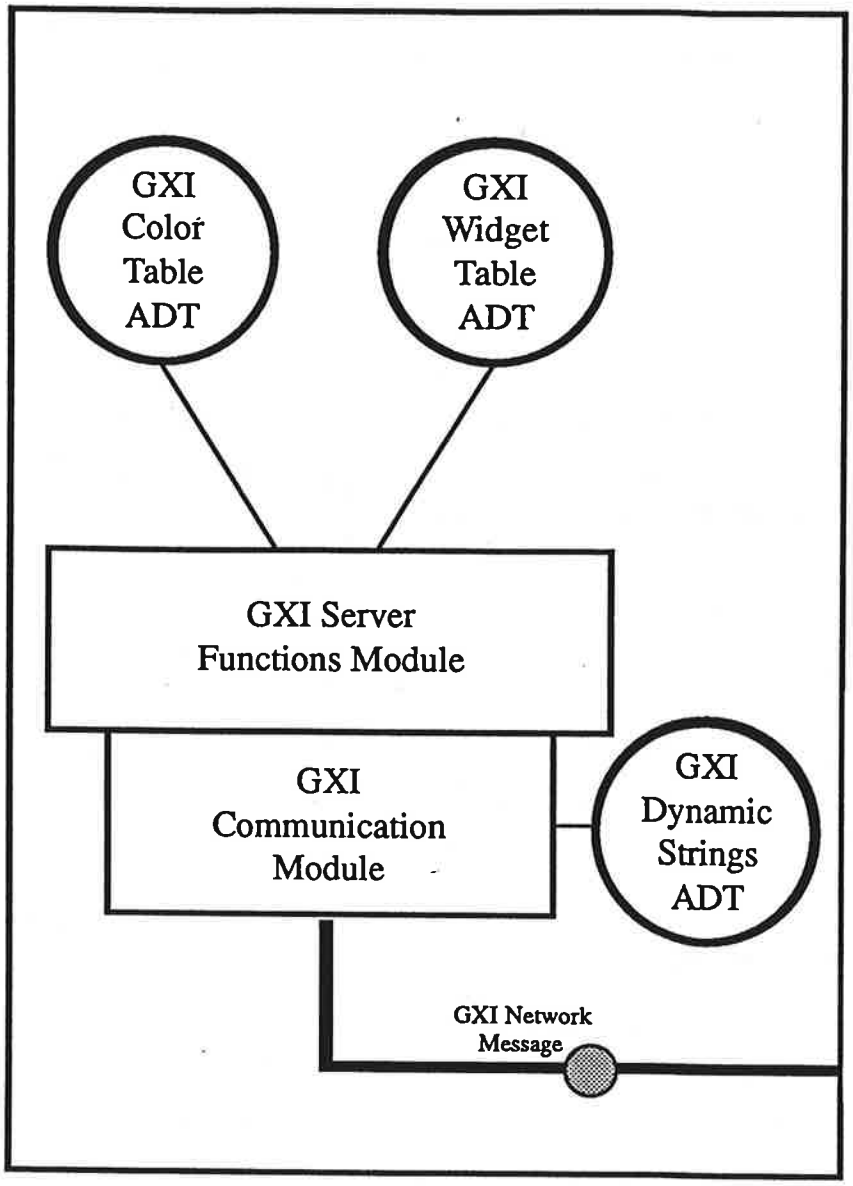


Figure 3.4: GXI Server Architecture

3.3.3 The Client Again

Once the client has successfully received the server's reply message, it proceeds to extract the task results. These results are then passed back to the GXI Client Functions module. At this point, the task results are passed either directly to the client in the case of a client written in 'C', or via the GXI CLIPS Functions module in the case of a client written in CLIPS. Now the cycle is complete and the client is free to execute its next request.

3.4 Abstract Data Types (ADT)

In an effort to assist GXI in the management of its environment, several abstract data types (ADTs) were developed. The specific purpose of each ADT ranges from allowing dynamic data transfer between the client and the server to increasing graphical performance.

3.4.1 Strings ADT

The purpose of the Strings ADT is to allow any number of logical character strings to be passed across the network from the client to the server and vice versa. This task is performed by concatenating all of the strings together into a single stream of bytes. To preserve the individual strings, each logical string is separated by a separator character. The process receiving this stream of bytes from the network simply calls the Strings 'Unpack()' function which separates the message back into its original 2-D (array of characters) format.

3.4.2 Color Table ADT

Allocating named colors in the X-Window environment is an extremely time consumming process. Considering the number of times a new color must be allocated, it was necessary to develop a method that would allow for a fast conversion of named colors to corresponding pixel values. The solution was a color hashing table ADT. For each individual client a color hashing table is created containing all of the color pixel values that the client wishes to use. The keys into the table are the logical color names themselves. To indicate to the server which colors the client wishes to use, it simply places the color names into an ASCII file and passes the file name to the server during its connection phase. Since the number of colors that a particular client may wish to use is typically quite small, the Color Table performs at a very high level producing almost no collisions.

3.4.3 Widget Table ADT

The Widget Table is perhaps the most important ADT used by GXI. The GXI server uses this table to keep track of all objects created by the client. The Widget Table entry itself exists as a union structure of the two main types of graphic objects available to the client (i.e., Forms and Menus). All other graphic objects can be stored in common fields residing just outside the union.

The first two of these fields hold unique object ID numbers for the specific object and its parent, respectively. Since X-Window manages its objects in terms of a hierarchical tree the object's heritage must also be reflected in the Widget Table. For example,

when the client wishes to destroy a specific object, all of its children must also be destroyed. Even though X-Window is responsible for performing this deletion on the actual children objects, it is the task of the Widget Table to perform this maintenance in its own environment.

Apart from the issue of maintenance, the Widget Table object ID value plays another important role in the GXI environment. It is this ID value which provides the link between the server's physical objects and the client's logical objects. When the client wishes to create an object, the ID value provides the handle to the specific object and is passed back to the client. From that moment on, the client references that object by using this unique object ID number. However, from the client's point of view, this object ID value is in fact the actual object.

The third field consists of a pointer to the actual object. This value is necessary for the GXI server to gain access to the object as it exists in the X-Window environment.

The fourth field of a table entry contains the basic type of the object. This is used as a tag into the union structure and provides a method whereby the Widget Table routines can quickly determine what kind of an object it is currently dealing with.

As explained earlier, the remaining fields exist as a union of Form and Menu records. The Form record holds the X-Window Graphics Context (GC) which is used to describe the specific Form's drawing environment. In addition, this record contains an exact graphical image of the current state of the Form's window, contained in an 'XImage' data structure that is used for GXI system maintenance on the specific Form's window. This task is described in more detail in Section 3.5.

As a result of these facilities the Widget Table is used extensively by the GXI Server Functions module in an effort to keep track of and manage the client's GXI environment throughout its session.

3.5 Image Restoration Techniques

The task of image restoration is an issue which all graphics programmers must deal with at one time or another. Image restoration is the task of redrawing a newly exposed region of the screen to give the appearance of graphical layers. In other words, users should be able to view the many graphics windows as existing in layers. The most prominent window would then appear to cover up the other windows with the least prominant window appearing to be deepest into the screen. A less prominant window may become exposed when its particular layer is shuffled up. The image previously obscured must now be redrawn in a transparent fashion to the user.

In an effort to assist its applications in allowing for image

restoration after exposure, X-Window offers a facility known as 'Backing Store'. The latter operates by allowing the client application to indicate that it wishes the contents of a certain window to be stored internally to X-Window. Therefore, when a section of the window becomes exposed, X-Window will perform an image refresh on the window thus preserving the client's graphical environment. However, in actual practice it became apparent that this facility is very much version dependent. Further, even when the 'Backing Store' facility is supported it is by no means a fail safe process. It is not uncommon for an exposed window section to be only partially restored thus compromising the integrity of the application. With this in mind, the development of a method which would provide X-Window implementation independency and also accurate image restoration became a necessity.

The solution to this problem utilized the X-Window 'XImage' data structure in addition to a few simple maintenance routines. The 'Xlmage' structure is used by the GXI server to store the graphics image of all client Forms. Therefore, when an exposure event occurs in a client window, the GXI server uses the corresponding 'XImage' structure to restore the exposed region. The image information itself is stored in the Widget Table which is easily accessed by the GXI maintenance routines. To improve performance, only the exposed region of the specific window is restored. Performing image maintenance in this fashion provides version independence in addition to fast and accurate results.

CAD Research Center, Cal Poly, San Luis Obispo, Jun.'95 [CADRU-10-95]

4. MMS: MERCURY Message System

In early 1993, with the completion of the AEDOT prototype for the US Department of Energy (Pohl et al. 1992) and the first ICODES prototype for the US Department of Defense (CADRC 1993), the CAD Research Center was forced to review the communication requirements of the ICDM framework. The increasing sophistication and scale of ICDM applications were placing more severe demands on the existing message-passing capabilities.

In particular, ICADS and AEDOT developers had consistently requested more communication facilities to be supported by GXI (Pohl 1992). Rather than duplicate communication facilities in different levels of software, or for different purposes, it seemed more appropriate to provide one message-passing system that could be used for all purposes. Even though the application developer could utilize socket code for specific purposes, it became clear that a general message-passing facility would provide a higher degree of integrity and reliability, or at least provide those features without redundancy. It was felt that it should be possible for developers of cooperative agents who may not be skilled UNIX system programmers to obtain general communication facilities by adding only a few simple calls to their code.

Furthermore, by providing a message-passing facility that would be capable of handling all communication needs, a desirable reduction in system resources should be achievable. Rather than have sockets within the blackboard coordination system, different sockets within GXI, and still different sockets for other specific communication needs, all communication should take place within a common system. This would reduce the number of buffers and communication processes running on each CPU.

Several commercial communication facilities available at that time were examined and rejected. Most distributed operating systems were either transaction oriented or devoid of utilities to support inter-process communication among the distributed processes. While they appeared to provide a high degree of reliability in executing the requested processes, or in automatically selecting the host CPUs to optimize the computation, they required the user to employ socket-level software within the processes in order to communicate with other processes. ISIS (Birman and Marzullo 1989) was an example of the former type of system and Plan 9 (Pike et al. 1990) was an example of the latter.

Furthermore, the message passing facilities in most distributed operating systems were so focussed on fail-safe delivery that the required protocols produced enormous overhead. With one exception they appeared to be too inefficient to be used in the highly interactive-reactive ICDM environment. This exception appeared to be the PVM (Parallel Virtual Machine) system (Sunderam 1990), which was discovered only midway through the development of the MERCURY

Message System (MMS) described in this Section. However, PVM later became the core of the Communication Management System (CMS) currently used by the CAD Research Center in the ICODES and FEAT applications (see Section 5).

In establishing the design objectives for a new communication and event management facility for the ICDM framework, the CAD Research Center adopted the philosophy that the facility should exist at both a logical and physical level. Logically, the facility should allow the clientele to communicate with each other through object-oriented messages. In other words, clients should be able to use any type of object as an inter-client message without transformation of its contents. Further, client application environments should be able to define and manipulate their own set of objects both statically and dynamically. This capability requires a high degree of flexibility within the interface presented by the communication facility. In addition to the issue of flexibility, clients should also be removed from any physical characteristics of their counterparts (Elmasri and Navathe 1989). This includes such characteristics as the current state of execution and the physical site location.

Another important issue deals with client authorization. Each application environment should be able to define and manage its own authorization of privileged facilities. This external authorization must work in conjunction with the underlying authentication and security mechanisms provided by the communication system.

Further, clients must be notified of pending events in a real-time fashion (Durfee 1988). The emphasis on real-time message-passing requires the elapsed time between the triggering of an event and the subsequent notification of the appropriate client(s) to be minimized. This requirement is crucial for providing data validity and currency within the application environment.

The need to accommodate applications that intend to manipulate both knowledge as well as standard numerical data, requires the underlying support system to offer its facilities in both a procedural and rule-based form (Pohl et al. 1988). Further, clients must be able to interact with one another independently of their implementation language. In this regard, rule-based clients should be able to transparently interact with procedural clients and vice versa.

Physically, all of the aforementioned facilities must be implemented in such a way as to provide for as high a degree of performance and application flexibility as possible. This requires the design of the communication system to seek out the most cost effective mechanisms for performing its assigned tasks. In addition, the selected mechanisms must be able to perform in a reliable and robust manner. For example, the underlying support system must handle such situations as receiving unknown message objects or the detection of an authorizer that is unknown to the application and may have malicious intent.

To accrue most if not all possible benefits of a networked workstation environment, it was considered important that the entire communication system be designed and implemented in a completely distributed fashion (Stevens 1990). Therefore the design was required to address such issues as site atomicity, fault tolerance, and agreement among distributed agents (Berstein, Hadzilacos and Goodman 1987). In addition, the system should support the dynamic allocation and deallocation of various resources within the application environment. This concept might even be taken to the point of encompassing entire sites.

4.1 Description of the MMS Implementation Design

It was decided that the MERCURY Message System (MMS) should take the form of a fully distributed, real time communication and event management system. To facilitate heterogeneous procedural and rule-based application environments its facilities should include mechanisms such as user-defined objects, user-defined and managed facility authorization, and real-time event notification.

Appendix B provides a description of the primary MMS functionality. To promote runtime efficiency, basic system utilities such as shared memory are utilized for local communication. Site-to-site communication takes place via TCP/IP based UNIX sockets (Stevens 1990). Further, MMS is designed in a fully distributed manner supporting such concepts as fault tolerance and site atomicity. Care was taken in the selection of its internal mechanisms to find a balance between efficiency and flexibility. The following sections outline the primary facilities provided by MMS.

4.1.1 Application Defined Message Types

MMS provides a mechanism for client applications to define their own message type protocols. This mechanism allows clients to attach special meaning to each message via the Message Type field located within the message header of each inter-client message. Clients may set up application specific message protocols allowing one application to trigger an event in another.

Clients wishing to add message types are required to edit a common MMS 'definitions' file that contains the various message types used throughout the MMS system. Apart from reserved MMS defined message types, these types are strictly client defined and can have any meaning attached to them that is appropriate to the application. Multiple message types may refer to the same physical message. However, any particular message type can only refer to a single physical message. Any MMS client may use an existing Client Defined Message Type or is free to define its own. In consonance with the design objectives, the notion of message type is included for purposes of providing a high degree of flexibility in an application environment.

4.1.2 Application Defined Objects

Many support facilities constrain users by providing only a small set of data type primitives with which its clients may interact. Applications wishing to interface with such restricted facilities are required to transform or translate their data objects into one, or several, of the available data type primitives.

MMS addresses this issue by introducing the concept of Client Defined Objects. Each application may define its own set of application specific data types along with a logical and concise set of access methods (Booch 1991). Further, these objects are not constricted to a single application. Instead, applications may share objects by passing them as inter-client messages.

Various guidelines and conventions are used by MMS clients to define a set of objects. First, the actual procedural syntax object structure (e.g., C language (Miller and Quilici 1986)) must be defined in a common MMS 'Client Defined Objects' file. Second, a 'C' syntax interface must be written which provides controlled access to instances of the specific object. This interface consists of two parts.

1. A functional definition of each interface function must be placed in an MMS 'Client Defined Object Interface Definition' file which can be freely accessed by any application designer. This definition provides a description of the object interface to prospective users.

2. The interface implementation must be written and entered into the MMS 'Client Defined Object Interface Implementation' file. In addition to creating and destroying instances of the specific object, this interface must contain a set of functions capable of setting and obtaining individual field values by name. However, this interface should not be concerned with providing direct access into sub-objects that it may contain. It is the responsibility of the sub-object's interface to perform this task. As an additional requirement, the interface must also contain a translator function capable of translating the contents of the specific object from its 'C' syntax form into an equivalent rule-based (e.g., CLIPS) form.

Once the appropriate file entries have been made, MMS must be recompiled to allow for the necessary update. Any application wishing to utilize the new object must be relinked with the current set of client defined object files. By taking a somewhat relaxed attitude toward application objects together with the employment of a set of strict interface definition requirements, MMS strives to achieve an environment rich in application flexibility, extendibility, and integrity.

4.1.3 Acting Client Authorizer Group

For reasons of data consistency and protection, there may be situations where it becomes desirable to limit the types of MMS actions any particular client is permitted to perform. However, to decide whether a client should be able to perform a specific action should be the sole responsibility of the designer of the particular application environment. MMS itself would have no way of determining exactly which clients should be allowed to perform which actions. For this reason the underlying support system should provide facilities whereby application environments can design and implement their own authorization protocols. Further, this facility must work in conjunction with the underlying authentication and security enforcement mechanisms provided by the particular support system.

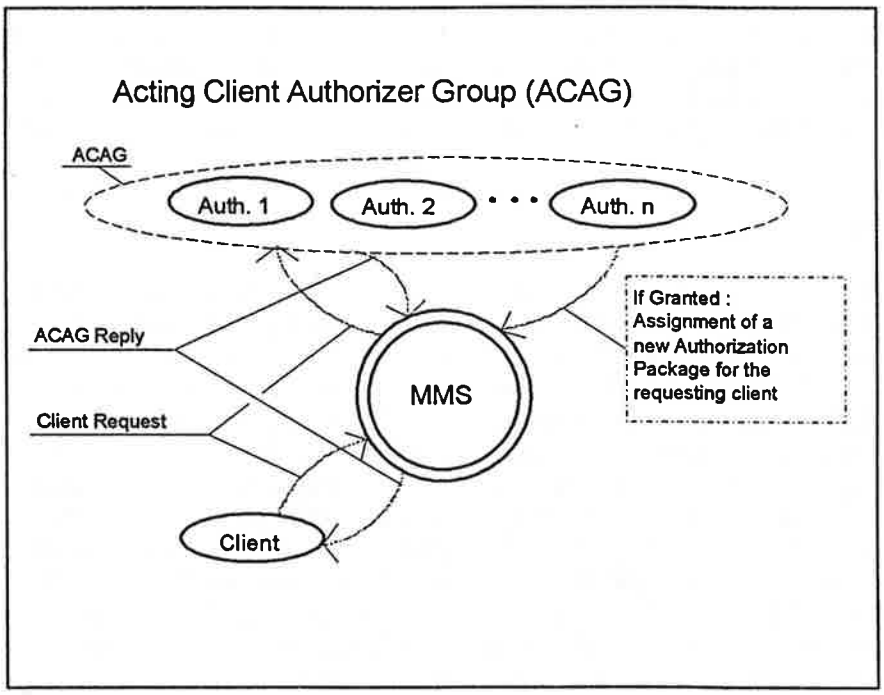


Figure 4.1: Acting Client Authorizer Group

With this requirement in mind MMS introduces the concept of an Acting Client Authorizer Group (ACAG). This group consists of a single, or multiple MMS clients designed in a manner that allows them to work in a collective fashion to determine the result of each client authorization request (Figure 4.1). MMS clients enter and exit this group through calls to 'MMSAssertAuthorizerStatus()' and 'MMSRetractAuthorizerStatus()', respectively. MMS associates an Authorization Package with each of its clients. Further, MMS provides facilities whereby clients can request an alteration in their Authorization Package. Each time a client attempts to perform

a 'protected' MMS action, the associated Authorization Package is checked to determine whether the particular client is, in fact, authorized to perform the desired action. Following, is a brief discussion of how client authorization requests are handled by the ACAG.

Each acting authorizer is notified of client authorization requests through the reception of a message of the type 'MMS_CLIENT_AUTHORIZATION_REQUEST'. This message is sent to the ACAG by the MMS each time a client requests an alteration of its current authorization status. Upon receiving a message of this type, the ACAG applies its particular application defined authorization protocol to determine whether the request should be granted. When a decision is reached, the ACAG notifies MMS as to its decision. This is performed through a call to the MMS function 'MMSAuthorizationRequestReply()'. In the case where the request is to be granted, the ACAG must take the additional step of assigning the new Authorization Package to the requesting client via a call to MMS. In either case, MMS notifies the requesting client of the decision concerning its request. As a final point, it should be noted that the requesting client has no knowledge that an ACAG even exists. The only interaction the client sees during this entire process is that which exists directly between itself and MMS.

4.2 The MMS Architecture

There are basically five components which together comprise MMS (Figure 4.2). Each of these components is designed with the concepts of atomicity, integrity, consistency, efficiency, and portability in mind. MMS is designed to work in a networked, UNIX workstation environment utilizing the TCP/IP protocol suite to provide site-to-site communication (Stevens 1990). In addition, the design of MMS employs the use of such fundamental UNIX mechanisms as shared memory and inter-process signals (Bach 1986). These basic tools were chosen over more powerful yet more machine dependent mechanisms in an effort to provide a system compatible with the concepts of portability and overall system integrity. The following sub-sections provide brief discussions of these five components fundamental to the overall architecture of MMS.

4.2.1 The Adjunct

The MMS Adjunct (MMA) constitutes the client interface to MMS. Clients wishing to utilize the facilities of MMS are required to 'link' their application to the MMA (Figure 4.2). All interactions between a client and MMS is handled via this adjunct. Further, it is the MMA that initially receives all communications directed at its associated client. Depending on the particular state of the client, the MMA will either trigger the appropriate client action or simply buffer the message for processing at a later time. In addition to providing the basic MMS interface, the MMA also contains a set of user-defined objects along with their interface.

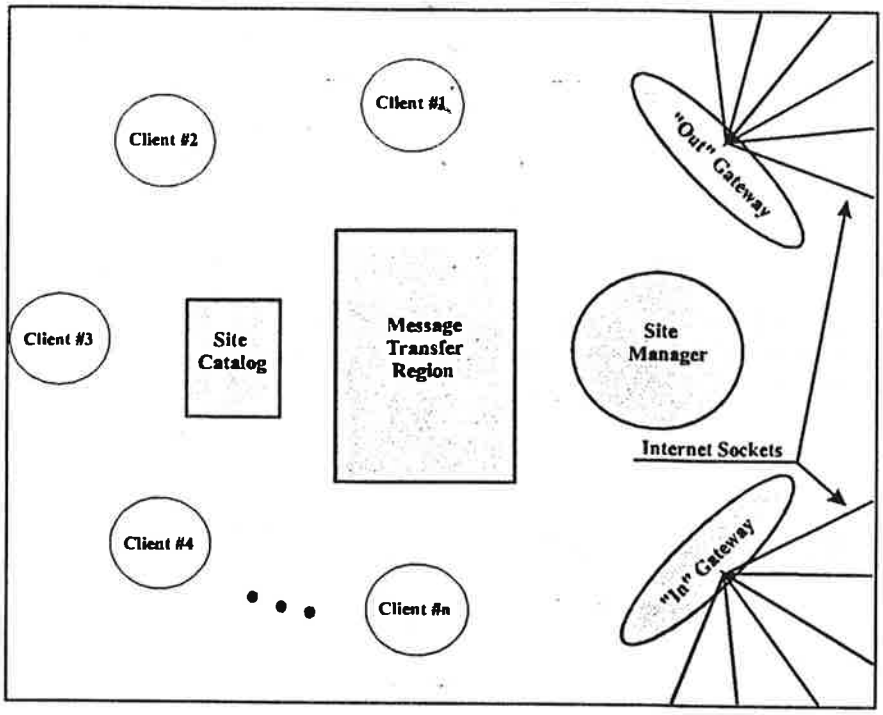


Figure 4.2: MMS Architecture

At this point it is appropriate to provide a brief example dealing with the MMA and the user-defined objects it contains. As mentioned earlier, any MMS client may create its own set of objects by placing a few entries in the appropriate MMS user files. Consider an application environment consisting of fifteen MMS clients. Further, ten of these clients are coded in the 'C' language (i.e., procedural clients) and the other five are coded in CLIPS (i.e., rule-based clients). Now suppose that for some reason a new communication protocol between two of the 'C' clients needs to be created. Further, this new protocol requires the creation of a new object which will be periodically passed between the two clients. The difficulty arises when one considers the actions which must be taken by each of the fifteen clients.

The most undesirable situation would be one where each client application would need to be recompiled, or at least relinked, to the new MMA which contains the new object. Considering the difficulty and potential problems with consistency and integrity with requiring all applications to take some sort of action simply because someone decided to define a new object, the disadvantages of this situation become quite evident. The most desirable situation would be one that would not require any action to be taken on the part of any client. In other words some mechanism should be provided that decodes, or deciphers message objects each time they are received. However, in practice the overhead required for a generic decoder to decipher each, or a sub-set of all message

objects would be simply too costly. Further, the only way this scheme would really work would be if there existed a basic set of fundamental data types that all other complex data types are comprised of. At best the decoder could decompose a particular message object into its fundamental data types which it could then pass to its client. In addition, one must keep in mind the fact that for any real processing of an object, there must exist a section of code which understands exactly what type of information the object contains. To process an unknown object without any high level knowledge of its contents would require that the processing take place at the fundamental data type level. This method is completely inconsistent with the notions of abstract data types and object oriented application design (Booch 1991). Considering these serious disadvantages, it is clear that a generic decoder approach falls far short in a practical arena.

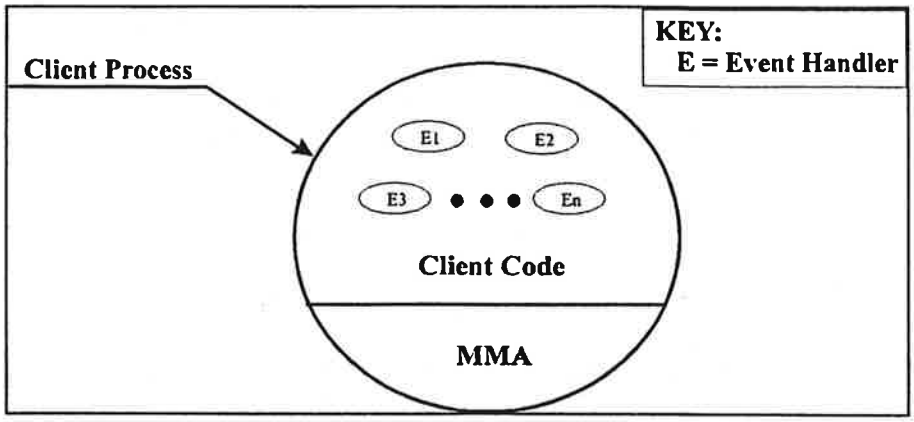


Figure 4.3: Client Process Architecture

With this in mind, MMS finds its solution on ground lying somewhat between these two approaches. MMS requires only clients who are particularly interested in processing a newly defined object to take some course of action. Interested clients are required to relink themselves with the appropriate MMA. Considering that they must recompile their application as a result of adding code to process the new object, this requirement is well within reason. In the example discussed above where a protocol between two clients required the addition of a new object, only these two clients would need to be aware of the addition. The other thirteen clients who are not directly affected by the newly defined object are not required to take any action. If by chance the MMA of an application receives an unknown object in a message, the object is simply discarded with no resulting ill effects.

It should be pointed out that the above discussion deals solely with application languages requiring compilation. Since CLIPS exists as an interpreter, a similar, yet somewhat more flexible approach can be employed (NASA 1989). It is possible for a programmer to extend a CLIPS application to process a new object by inserting additional rules into the knowledgebase without the need

for recompilation. In this case MMS still requires that the particular CLIPS Expert System Shell relink itself with the appropriate MMA. However, no recompilation is required since the actual 'C' code of the application did not change.

4.2.2 Shared Memory Manager

The Shared Memory Manager was designed to provide multi-process applications with the ability to create and manipulate shareable resources. Existing as self-managing linked lists of shared memory pages, applications increase or decrease the size of a shared resource by allocating or deallocating pages of memory. Once allocated, the caller is free to format the new page as desired. The integrity of a shared resource is maintained through the use of a Lock Manager. The Lock Manager allows a single Shared Memory List page to be locked in either an 'exclusive' mode for updates or a 'shared' mode for reading. This approach allows a Shared Memory List to be accessed by several clients simultaneously without jeopardizing the integrity of its contents. In fact, in the case of parallel reads, a single Shared Memory List page may be accessed by multiple clients simultaneously. Figure 4.4 provides an illustration of the architecture of such a Shared Memory List.

4.2.3 Local Site Catalog

The MMS Local Site Catalog (LSC) contains a variety of information describing the various clients, objects, and resources known to the particular MMS site. Physically, the LSC exists as a Shared Memory List that can by shared among multiple MMS clients on the local site (Stevens 1990). Each MMS site contains its own LSC. The particular state this shared resource is in at a specific point in time determines the type of access that is permitted. For example, for data consistency reasons a 'Read' access request is not permitted while the LSC is in an 'Update' state. Access to the LSC is controlled via the acquisition of two types of locks, Shared and Exclusive. Shared locks are associated with 'Read' operations and Exclusive locks are associated with 'Update' operations. Since 'Read' operations do not physically alter the contents of the LSC they can be performed in parallel, thus the name Shared lock. However, 'Update' operations physically alter the LSC's contents and consequently require an Exclusive lock. Controlling access to the shared resources ensures the integrity and consistency of its contents relative to 'Read' and 'Update' operations (King and Collmeyer 1973).

As mentioned earlier, the LSC provides descriptions of various entities and resources known to the particular site at any given point in time. It should be noted that LSCs existing on separate MMS sites may differ significantly in their contents. This is a direct result of the employment of site atomicity. Each particular site has its own view of the world. In other words, any given site should know about entities and resources that it has an interest

in. Typically this information consists of entities and resources located on the local site in addition to a collection of those located on foreign sites. The integrity of the contents of a LSC should be limited to the confines of the local site. On the other hand, the integrity of LSC information about foreign sites is not enforced. Rather, this information provides an approximate description only of foreign entities and resources.

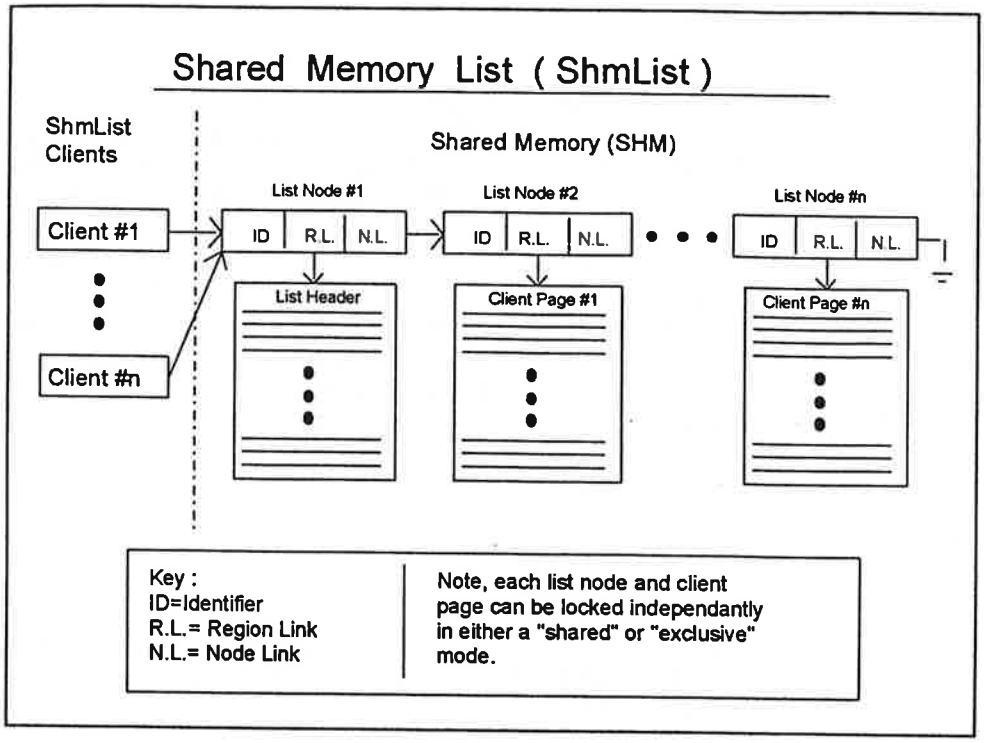


Figure 4.4: Shared Memory List

Consequently, the foreign site information must be verified if it is intended to be used. This verification may take the form of querying the foreign site directly to determine whether the information is still valid. However, to perpetually insure the integrity of foreign information located in a LSC would require a tight coupling between sites. This coupling would greatly reduce the degree of atomicity of the site, as well as accruing serious overhead in the form of additional inter-site communication. Instead, MMS provides mechanisms for one site to query another site. If a certain site wishes to obtain information about something that is not located on its particular site it has the ability to broadcast an information request to all, or a sub-set, of known sites asking if anyone knows about the entity or resource. If the requesting site receives an acknowledgment from any of the other sites, it utilizes the new information and consequently makes a note of it via an entry into its LSC. Later, if the initial site needs to reuse this information it can locate the appropriate entry in its LSC with the understanding that it may no longer be valid and may require additional verification. Employing a more relaxed

attitude toward foreign information integrity in conjunction with deferred data verification results in a system which reaps the numerous performance and atomicity benefits inherent in fully distributed environments.

4.2.4 Local Data Transfer Region

Similar to the LSC, the Local Data Transfer Region (LDTR) exists as a Shared Memory List that can be shared among the various MMS entities. Access to the LDTR is determined in the same manner as the LSC. The main function of the LDTR is to provide a common 'mailbox' into which MMS agents can place or from which they can retrieve inter-client messages. Clients wishing to send an inter-client message acquire 'Send' access to the LDTR and place their message in the appropriate mail slot. Receiving clients acquire 'Retrieve' access to the LDTR and simply pick up their mail. It should be noted that 'Sending' and 'Retrieving', as used in this example, is handled directly by the MMA of the particular client transparent to the associated client application.

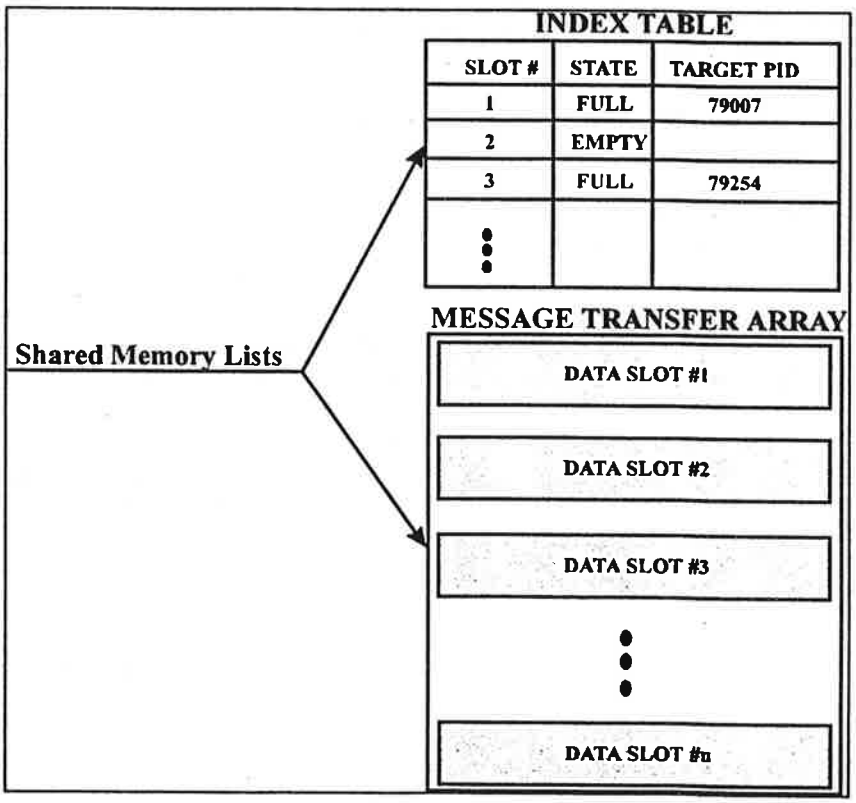


Figure 4.5: Message Transfer Region

A closer look at the LDTR reveals a somewhat complex physical structure (Figure 4.5). The LDTR basically comprises two tables. The first table can be thought of as an index table. The purpose of this table is to map symbolic client names to the slot(s) that

contain their message(s). In addition, this index table indicates the overall state of the LDTR (i.e., which slots are occupied and for whom). When an outgoing message is placed into the LDTR the MMA of the sending client performs the following functions.

First the MMA searches the index table for an available slot(s). If a slot is not found, the MMA releases its 'Send' access to the LDTR and proceeds to the end of the 'Send' access queue for that resource. In the case where a slot(s) is found, the MMA marks the entry as occupied and places the ID# of the target client into the appropriate slot field. The MMA then uses the index value of the index table slot as a direct mapping into the second table of the LDTR.

The second table of the LDTR is known as the Message Transfer Array (MTA). This array is the evaluation where the actual message objects being transferred are placed. In the initial implementation of MMS, the size of each array element was set to some predetermined value. This results in the possibility that large message objects may span several array elements. Each array element contains a header indicating the size and sequence number of its message fragment. In addition, this header also contains a field indicating the array element containing the next, if any, fragment of the particular message object.

It should be noted that this approach does have a serious disadvantage. Since the size of each MTA element needs to be determined at compile time, the overall efficiency and performance of inter-client message passing can be greatly affected by this decision. It is possible to make the size of each MTA element a function of the average object size, for example. However, this provides no estimation of the size of the objects that will actually be passed at run time. In the case where only small message objects are being passed it is possible that much of the space allocated to the MTA will be wasted. This unused portion of each MTA element could have been used to increase the overall number of available data transfer slots. Further, the overhead associated with spanning large message objects over multiple MTA elements could seriously decrease the overall speed at which message objects are transferred from client to client. While this was considered acceptable for the initial implementation of MMS, it was noted that later versions should contain variable length MTS elements that have the ability to grow or shrink dynamically according to the size of the message object they contain.

4.2.5 Local In/Out Gateways

Local In/Out Gateways can be thought of as portholes to the outside world. Each MMS site is connected to all other MMS sites via these gateways. At the physical level, each Out Gateway has a direct Internet socket connection to the In Gateways of all other MMS sites (Stevens 1990). Conversely, each In Gateway has a direct connection to the Out Gateways of all other MMS sites. Therefore,

if a message must move between MMS sites it is guaranteed that only a single site-to-site transfer is required. In other words, no intermediate sites will need to be visited during a message's journey between the source and destination sites. Further, in the case where a new site is dynamically allocated, or an existing site is deallocated, the number of gateway connections will be altered accordingly.

In an effort to simplify the design of MMS, the functionality of these gateways was designed to be similar to any other MMS client. In fact, each gateway itself exists as a client to MMS and is listed as such in the LSC. As clients to MMS, a gateway has the potential to both send and receive inter-client messages.

At this point a distinction needs to be made between the different types of inter-client communication. In the first type both the sending client and the receiving client are located on the same site. In this case, the message object moves from the sender's MMA to the receiver's MMA via the LDTR. Since the message never leaves the local site, the gateways are not utilized.

In the second type of inter-client communication the sender and the receiver are not located on the same MMS site. In this case the inter-client message must pass between MMS sites via the MMS gateways. In the later case, the receiving client on the sender's site is, in fact, the local Out Gateway. Notification and reception of the inter-client message by the local Out Gateway is performed in exactly the same manner as with any other MMS client. The difference occurs in the processing of the particular message. Out Gateways process inter-client messages by routing them to the In Gateway of the appropriate foreign site. Once the In Gateway of the receiving site receives the inter-client message it in turn acts as a standard MMS client and performs local inter-client communication to the original destination client. It should be noted that the use of the MMS gateways in addition to the fact that the destination client need not be on the sender's site is completely transparent to the sender and receiver of the inter-client message.

4.2.6 Local Site Manager

As the name implies, the Local Site Manager (LSM) is responsible for managing the various MMS resources and entities located on a particular site. In addition, the LSM represents MMS as the spokesperson for its particular site. Any communication directed specifically to a MMS site is handled by the associated LSM. For example, in the case where a new MMS site is to be allocated in the system, MMS will execute a series of protocols requiring internal communication between MMS sites. This communication will typically take place between LSMs located on the various MMS sites. Again it should be pointed out that MMS clients never specifically address a particular LSM. Rather, any direct communication a client has with MMS is addressed to the MMA associated with that client. The MMA

then determines which site to direct the request to in a transparent fashion to the issuing client. Further, even the MMA only goes as far as determining which site to relay the request to and makes no attempt to directly address a LSM. Any incoming messages specifically directed to a MMS site will be passed to, and handled by, the LSM for that site.

4.2.7 Fact Manager

As previously mentioned, MMS clients may exist in either of two forms, procedural 'C' programs or rule-based CLIPS programs. Whereas procedural clients deal with objects implemented as 'C' structures, rule-based clients manipulate objects implemented as CLIPS facts. A fact is a chunk of information that can be asserted into the 'current scope of knowledge' (NASA 1989). The Fact Manager was developed to provide a common representation of fact objects across a heterogeneous client base. In other words, fact objects may exist in not only rule-based environments, but also in procedural environments. Therefore, there should be a common representation of a fact that can be passed between heterogeneous clients without concern of the client implementation language. In addition, the Fact Manager also provides 'C' clients with an interface capable of manipulating fact objects in a procedural environment. Appendix B provides a description of the functionality of the Fact Manager

4.3 Local and Site-to-Site Communication

As mentioned earlier, communications in MMS can be divided into two categories. The first category deals with inter-client communication between two MMS clients existing on the same site. For the purpose of the following description this type of communication will be referred to as local communication. The second category deals with communication between MMS clients existing on different sites. This type of communication will be referred to as inter-site communication.

4.3.1 Local Communication

The information flow underlying local communication within MMS is shown in Figure 4.6.

A client initiates the sending of a message by calling the appropriate MMA function (i.e., MMSSendMsg(), MMSBroadcastMsg(), MMSSendAndAckMsg(), etc.). The MMA then reads the header information from the client message. This header contains such information as the size of the message object and the client who the message is directed to. In addition, the header also contains the particular Client Defined Message Type. As discussed previously, MMS makes no attempt to decipher this field. It is solely for the benefit of the sending and receiving client. The

only use that MMS makes of this field is as an index into the Client Message Handler Table (CMHT) which provides mappings of incoming messages to their associated actions.

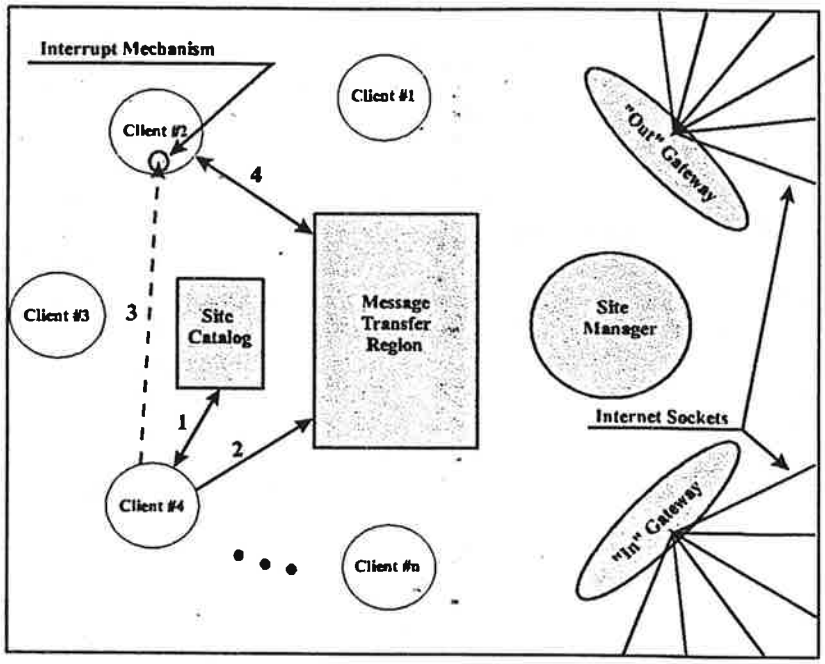


Figure 4.6: Local Communication

Once the appropriate information has been extracted from the message header, the MMA uses the symbolic name of the destination client to locate the appropriate client ID# in the LSC. This is illustrated as step 1 of Figure 4.6. The MMA then places the actual message object into the LDTR (step 2). If this step is successful, the MMA uses the destination client ID# to send a 'message pending' signal to the MMA of the destination client (step 3). This signal essentially interrupts, or preempts the destination client so that it will be able to handle the particular event in a real-time fashion. Depending upon which MMS send function was called, the sender's MMA may either wait for an acknowledgment from the destination MMA or simply return control back to the client application. When the destination MMA receives the 'message pending' signal, it takes action by attempting to retrieve its mail from the LDTR (step 4). If this step is successful, the destination MMA uses the Client Message Type located in the message header to map to the event handler that the client has previously associated with messages of this type.

If an entry for the particular message type is found, the MMA calls the client event handler passing the message object as an argument. If no entry is found, the MMA discards the message. It should be noted that since local communication executes solely within the realm of a single site, the In/Out Gateways, in addition to all other MMS sites, are not affected by any portion of the transaction.

4.3.2 Site-to-Site Communication

Figure 4.7 provides a step-by-step illustration of the actions taken by MMS on the sender's site to perform site-to-site communication. Figure 4.8 illustrates the actions taken by MMS on the receiver's site. Both of these illustrations will be referenced throughout the following discussion.

Access to the LSC by the sender's MMA in step 1 of Figure 4.7 reveals the fact that the destination client is not located on the same site as the sender. In this case the MMA of the sender extracts from the LSC the site reference of the destination site, if available, in addition to the ID# of the local Out Gateway. If an entry for the destination client does not exist in the LSC a system wide query would be issued in an effort to obtain the information. Considering the potentially large number of MMS sites that might compose the system as a whole, this is another area of concern with respect to overall system efficiency. One possible solution would be to encode the location of each client in a 'client identity' object that could be passed around and consequencially used to directly reference that client.

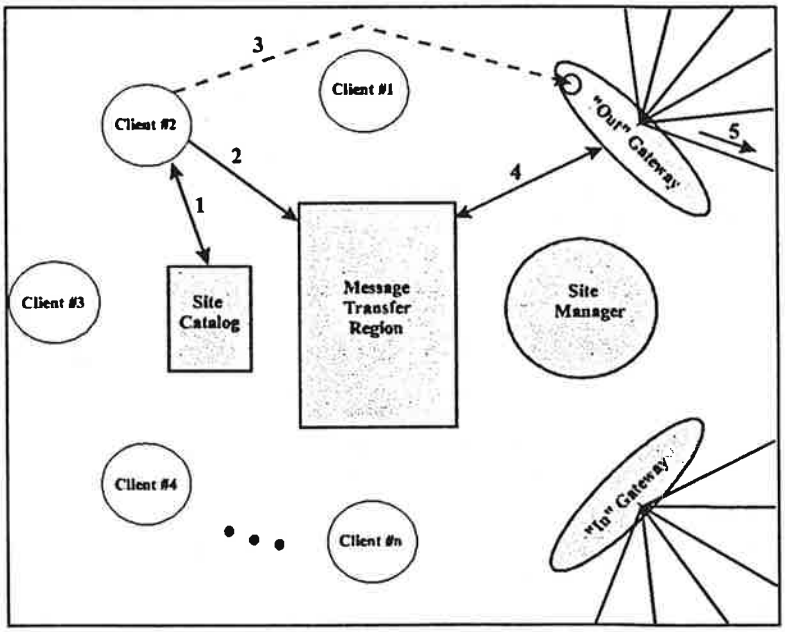


Figure 4.7: Site-To-Site Communication (Sender's Site)

The MMA then places the physical message object into the LDTR in the same manner as in step 2 of Figure 4.6. Step 3 of Figure 4.7 illustrates the sending of a 'message pending' signal to the local destination client. However, in this case the local destination client becomes the Out Gateway. Acting as any other MMS client, the MMA of the Out Gateway retrieves the message object from the LDTR

and passes it to the corresponding Client Event Handler (step 4). The Client Event Handler processes this message by sending it to the In Gateway of the destination site (step 5). At this point the message object has physically left the local MMS site enroute to the destination site.

Once the client message enters the destination site it is picked up by the local In Gateway of that site (step 1 of Figure 4.8). The In Gateway then processes this message by performing local client-to-client communication of the message object to the original destination client. Steps 2, 3, 4, and 5 of Figure 4.8 are identical to steps 1, 2, 3, and 4 of Figure 4.6. The destination client has absolutely no idea that the client who sent it the message exists on a foreign site. In terms of client-to-client communication, MMS presents its clients with a view of the world that is completely independent of physical location. As indicated earlier, this independence is essential if the concept of atomicity is to be preserved.

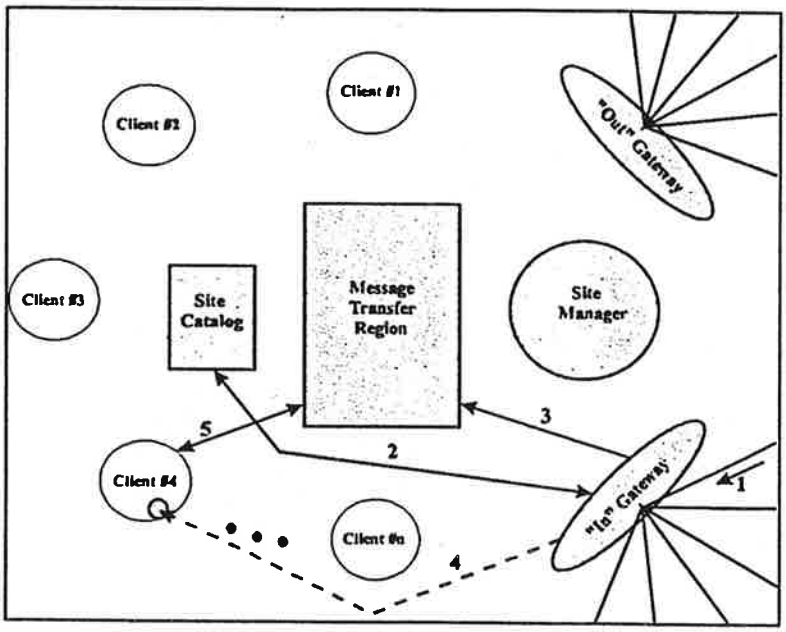


Figure 4.8: Site-To-Site Communication (Receiver's Site)

4.4 Possible Extensions and Enhancements

Under certain circumstances the LDTR can be expected to become a source of congestion for client-to-client communication. With this in mind, it is essential that the availability of an empty slot in the LDTR be as likely as possible. To create this situation, the LDTR could be enhanced to allow for dynamic slot growth during execution. This growth should occur both at the individual MTA slot level in addition to the number of MTA slots as a whole.

Since the LDTR exists as a sharable section of memory this region can be either increased or decreased utilizing standard UNIX system calls (Bach 1986). However, the internal structure of the LDTR must provide sufficient information at the MTA slot level to facilitate this dynamic growth. The information may take the form of a series of headers which are associated with various components of the LDTR. A general header could be associated with the LDTR as a whole. This header would indicate the current number of entries in both the LDTR Index Table in addition to the MTA. An additional header could be placed at the beginning of each Index Table entry indicating the current location (address) and size of the associated MTA element. In this manner the current state of each MTA element would be fully described in the associated LDTR Index Table entry.

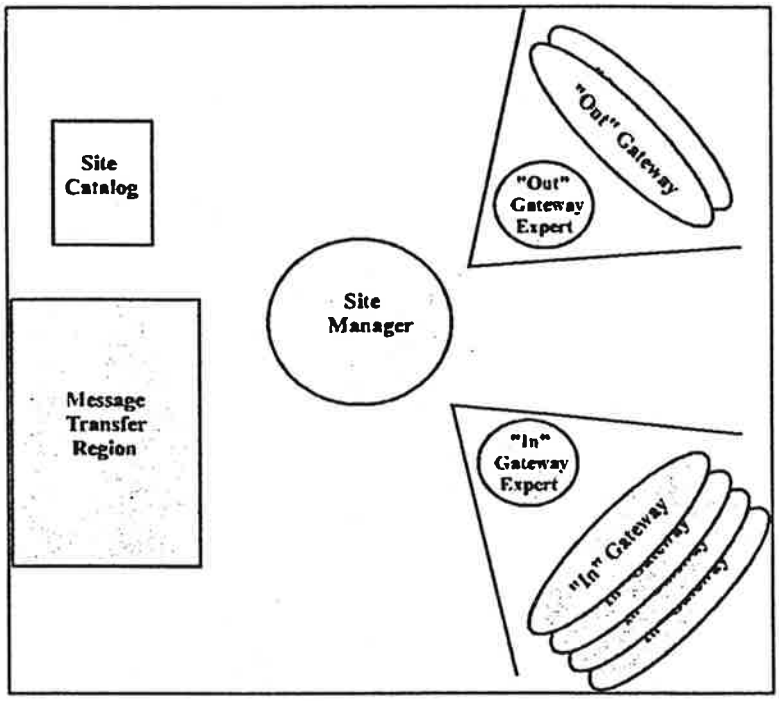


Figure 4.9: Multiple In/Out Gateways

Another possible source of congestion during inter-client communication exists at the In\Out Gateway level. In the current GXI implementation, all outgoing messages must pass through the Out Gateway of the sender's site. Similarly, all incoming messages are required to pass through the In Gateway of the destination site. Since these Gateways are the only mechanisms through which messages can leave or enter a particular MMS site, they must be designed to optimize performance.

The solution to this problem deals with the desire to always have a Gateway available when a message is to enter or leave a site. The approach proposed is to implement the notion of a Gateway server.

This server would be capable of spawning additional Gateways depending on the current state of Gateway congestion. Therefore, the number of Gateways existing within the particular MMS site at any point in time could be dynamically increased or decreased depending on need. Further, the decision of exactly when to spawn an additional, or terminate an existing Gateway would be performed by a Gateway Expert. Each Gateway set would be managed by one of these Gateway Experts. Periodically, each Gateway Expert would evaluate the current state of congestion within its particular set of Gateways. If the particular Expert determines that a change in the current number of Gateways is necessary, it will either allocate, or deallocate a Gateway. Figure 4.9 provides an illustration of this approach to Gateway design. Note that all other components of the MMS site would be essentially unaffected by this design enhancement.

CAD Research Center, Cal Poly, San Luis Obispo, Jun.'95 [CADRU-10-95]

5. CMS: A PVM-Based Communication Facility

With the completion of the ICODES (Integrated Computerized Deployment System) prototype for the Department of Defense in early 1994 (CADRC 1993) the CAD Research Center intensified its exploration of commercially available and public domain inter-process communication systems. Now that the ICODES project was about to enter a second stage leading to an end-user product, it appeared desirable for the CAD Research Center to focus its efforts on the cooperative aspects of multi-agent applications rather than the underlying communication facilities. In any case, the developing industry and government interest in distributed, cooperative systems had precipitated a significant amount of work by universities and government research laboratories directed toward the development of message-passing systems (Flowers et al. 1991, Rosenberry et al. 1992, Butler and Lusk 1992, Gropp and Smith 1993, Turcotte 1993, MPI 1994, Gropp et al. 1994).

PVM (Parallel Virtual Machine) which grew out of the Heterogeneous Network Computing research project undertaken jointly by Oak Ridge National Laboratory, University of Tennessee, Emory University, and Carnegie Mellon University, attracted our attention and was eventually selected as the core of CMS (Communication Management System) the most recent ICDM message-passing facility (Geist et al. 1993). CMS, as described in this Section, provides an object-based layer above PVM with interrupt capabilities.

5.1 Parallel Virtual Machine (PVM)

PVM exists as a public domain communication system supporting configuration, initiation, monitoring, communication, and termination of UNIX processes across a heterogeneous network. The goal of PVM is to present its users with a single, comprehensive virtual machine on which all communication takes place. Dynamic in nature, PVM allows users to configure a variable number of machines into this virtual environment. PVM configures additional machines by installing a PVM deamon on the new site. It is the task of these deamons to effectively manage their respective sites. These management responsibilities range from maintaining the integrity of the local PVM environment to routing inter-client communication.

PVM clients communicate with each other by packing message components, such as integers and strings into dynamic PVM buffers. It should be noted that this decomposition of high level objects into basic data types is incompatible with the notions of object-oriented communication. This is a deficiency that is present in many currently available communication systems, supporting heterogeneous network environments.

Clients may send these buffers to other clients in either a point-to-point or broadcast fashion. The buffers are then passed transparently to the local PVM deamon which in turn routes them to

the appropriate receiving client(s) (Figure 5.1). The receiving clients are required to query PVM in order to ascertain whether any pending communication exists. This query can be filtered to allow clients to obtain messages of a specific type from a specific client. In addition, receivers may be either put to sleep until the requested message arrives or may return immediately (i.e., empty handed) to allow the caller to continue with the next action. Once a buffer is received it must be parsed by the receiver, and its contents must be unpacked. This assumes that the receiver knows the precise format of the buffer. The requirement for the receiver to have this knowledge presents potentially significant difficulties in highly dynamic environments.

Once the low level components have been successfully unpacked the initial object can be reconstructed. This somewhat laborious decomposition and reconstruction process is necessary due to the fact that data types, such as integer and floating point values, are often represented differently in different computers.

Not withstanding these inherent deficiencies, PVM does strongly support the notions of fault tolerance, automicity, and distributed application environments. Therefore, it was selected to form the underlying 'toolbox' for the CMS communication facility.

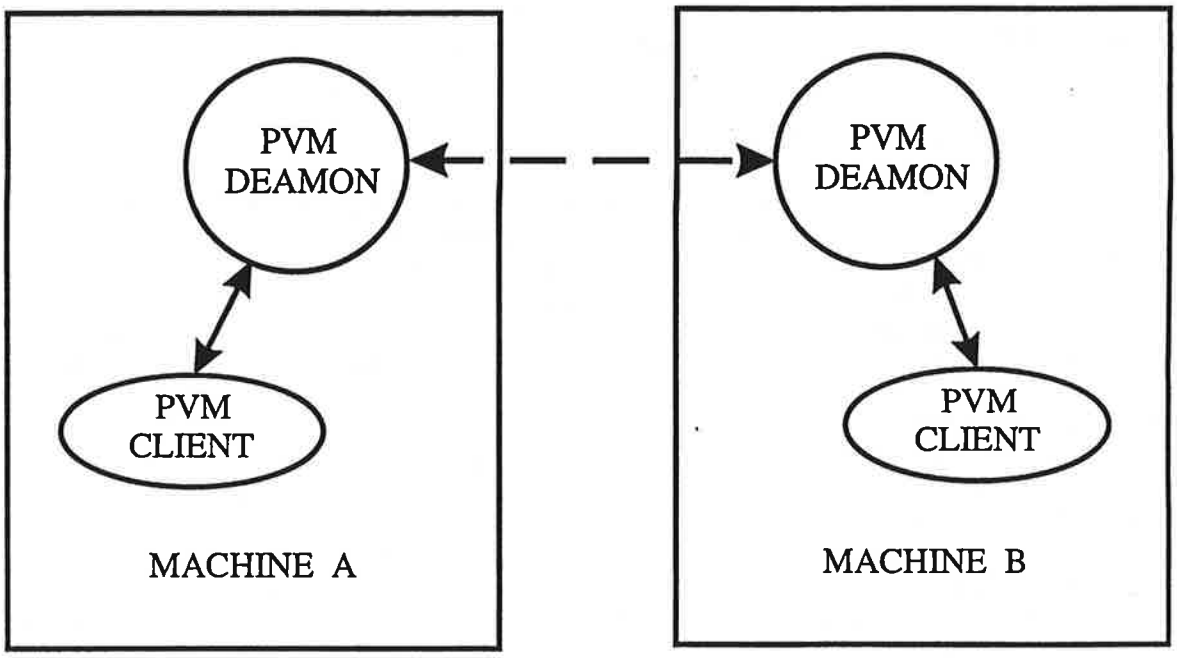


Figure 5.1: PVM architecture

5.2 CMS Sessions

CMS configures its current client base into a 'CMS Session'. Dynamic in nature, the population of a CMS Session may grow or

shrink as clients enter or exit the application environment. A CMS Session may involve any number of procedural or rule-based clients existing across a heterogeneous network of machines. CMS allows its clients to interact with one another in a seamless fashion. In other words, CMS clients may view their application environment as a collection of homogeneous components existing on a single, comprehensive virtual machine. Further, CMS clients do not require knowledge of either the physical location or program anatomy of their counterparts to communicate objects.

Clients enter a CMS Session through a call to 'CMSConnect()', indicating their logical name. Upon successful connection, 'CMSConnect()' returns the CMS representation of the caller. It is this CMS 'identity' object that allows clients to directly address each other within the confines of a CMS Session. Once connected the caller is free to perform a variety of CMS functions. Appendix D provides concise descriptions of the primary functionality available to CMS clients.

Of particular interest among these functions is the ability for clients to join CMS groups. Clients are able to enter and exit any number of application defined groups throughout the course of execution. These groups are primarily used during broadcast communications. All CMS broadcast functions accept a logical group name that is used to determine which clients should receive the particular broadcast communication. It is common practice in CMS applications for a potential client to connect to the current Session, join a 'system' group, and then broadcast its identity along with a brief description of its abilities to all members of the group. Upon receiving the client's 'identity' object interested members of the group will reply with their own 'identity' object, thus allowing for direct, point-to-point communication between clients.

This handshake protocol is essentially analogous to a stranger entering a room full of people, and indicating his or her name and intentions. Anyone who is interested in carrying on a dialog with the stranger would in turn indicate their name and intention, thereby effectively incorporating the new member into the course of discussion. Figure 5.2 illustrates the CMS equivalent of this common human situation.

5.3 Creation and Manipulation of Remote Objects

CMS clients are not limited to constructing and manipulating objects within their own local process environment, but also have the ability to create, modify, and delete objects in remote client environments. This can be accomplished through calls to 'CMSCreateRmtObj()', 'CMSUpdateRmtObj()', and 'CMSDeleteRmtObj()', respectively. Each of these functions takes in a CMS object description which is used to represent a specific instance of a particular class. This description contains both a system-wide unique instance identification (ID) and a dynamic list of object

slot name and slot value pairs. For the creation of a remote object this list defines a set of slot initializations that are required to be performed on the newly constructed object. In the case of an update, the list indicates only the names and values of those slots that are desired to be changed. Unlike creation and updates operations, remote object deletions require only the instance ID of the target object since no slot manipulation is intended.

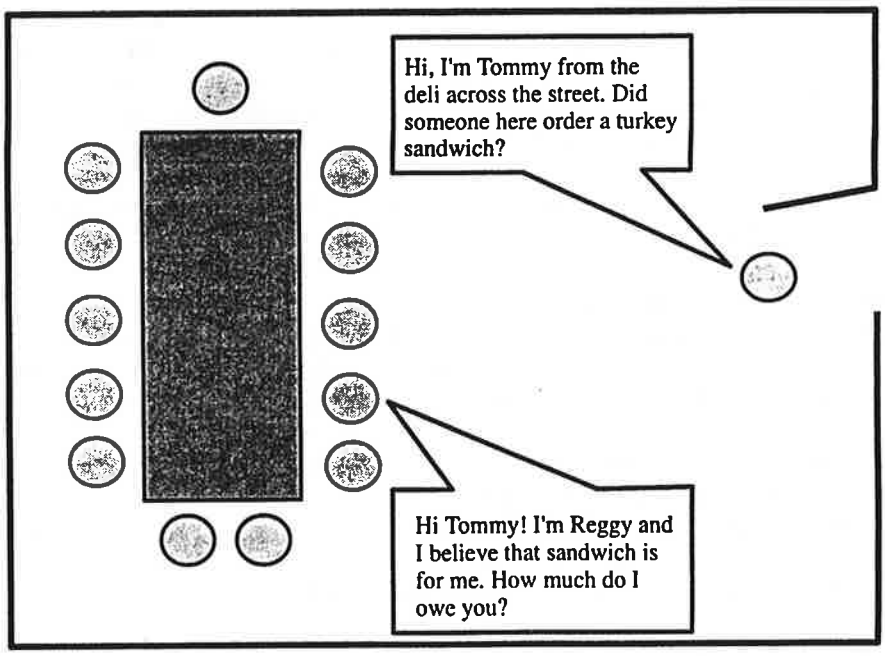


Figure 5.2: The CMS handshake protocol

By allowing objects to be created and manipulated remotely, in the same manner as local objects, CMS allows applications to view the communication of object-based information as a series of constructions, attribute sets, and destructions. This is in direct compliance with the object-oriented paradigm. Figure 5.3 illustrates the construction, manipulation, and destruction of a remote object.

5.4 Real-Time Event Management

The manner in which clients are notified of pending communications is a critical factor in determining the overall performance and efficiency of a CMS application. Issues such as CPU usage and information currency depend directly on how clients are notified of pending events. Many currently available communication facilities, including PVM, require that clients either repeatedly query for pending messages or that they be put to sleep until a pending message exists.

While both of these methods are desirable in some situations, each has its distinct disadvantages. The costs associated with repeatedly asking the communication facility for pending messages can be substantial in multi-process environments. In order to repeatedly perform such a query, the client is required to take up valuable CPU cycles that could be better utilized by other processes. An extension of this approach involves the use of an alarm mechanism that periodically wakes up the receiver allowing it to check for newly arrived messages. However, since the occurrence of events is dynamic in nature it is difficult to project an appropriate static time interval between future incoming events. In practice, this extension leads to a solution with consequences that may or may not be desirable.

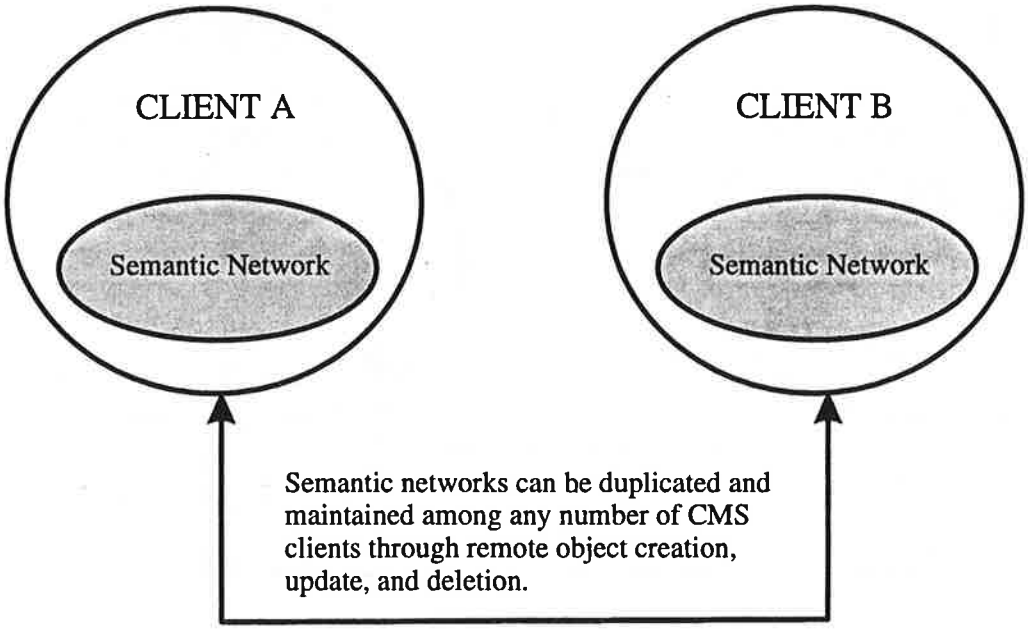


Figure 5.3: Construction, manipulation, and destruction of a remote object

To solve this dilemma, CMS takes a very different approach. The methods of event notification mentioned above view event notification from the receiver's point of view (i.e., when should a client initiate a query to check on pending events). The obvious answer to the question: "When should I ask for the next event?" is of course, "When it occurs!". CMS clients do not ask for events. When events occur, CMS client are immediately notified through process preemption. The CMS component of the sender's process preempts the receiver(s) after sending the particular message(s). Once preempted, the CMS component of the receiver's process formally receives the message(s) and passes it to an event handler that has been preregistered by the receiver (Figure 5.4).

Clients can protect 'critical sections' of their code from

interruption by temporarily disabling this transparent reception mechanism. Such 'blocked' events are simply buffered by CMS until preemption is reenabled. As a result CMS clients do not waste time asking for events that may not exist. However, to accommodate situations where an explicit receive is appropriate, CMS also provides various 'receive' functions in both a 'blocking' and 'non-blocking' mode. All CMS receive states, whether explicit or implicit, allow the caller to specify a particular message type and a specific sender. However, a wildcard format is available for either of these two parameters. By implementing event notification in this manner, CMS offers its clients an environment rich in application flexibility and efficiency.

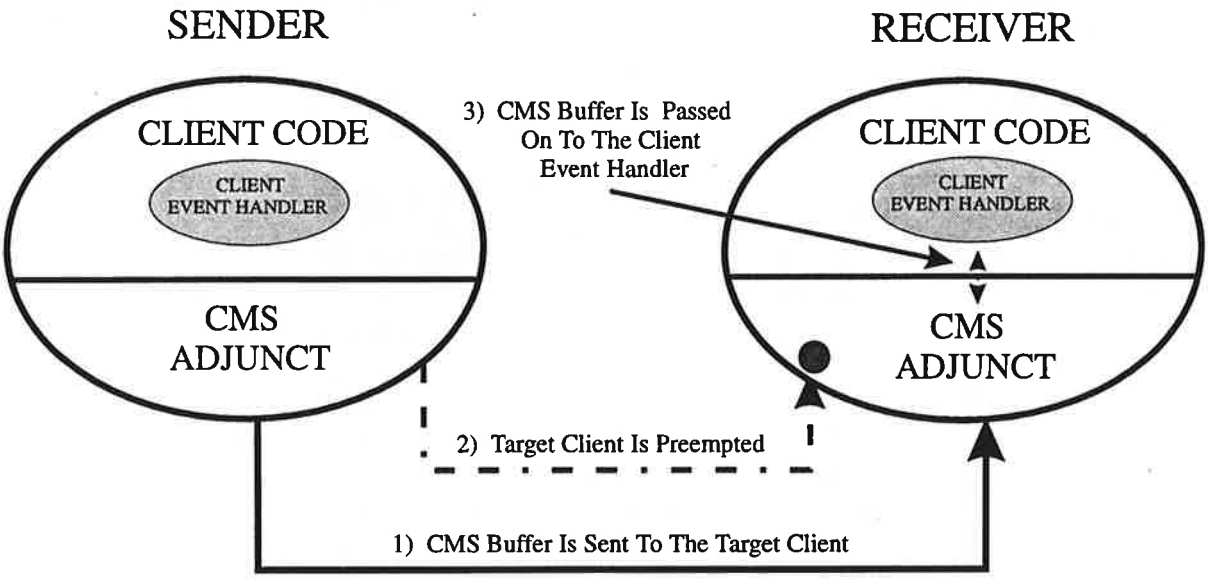


Figure 5.4: Event notification through preemption

5.5 The CMS Architecture

At the most fundamental level, CMS is designed to interface with procedural clients written in the 'C' programming language (Miller and Quilici 1986). CMS implements this functionality by providing its clients with a CMS library consisting of three principal components: a Session Manager; a Buffer Manager; and, an Object Manager (Figure 5.5). To extend CMS to interface with rule-based and more formal object-oriented clients, two additional components have been incorporated into the design; namely, the CLIPS Interface and the C++ Interface (NASA 1991, Stroustrup 1986). Both of these extensions are discussed in Section 5.6 under the heading "Object Interface Architecture".

5.5.1 Session Manager

The Session Manager is responsible for and provides the basic functionality of any given CMS client session. This functionality includes services such as: client connection and disconnection; broadcast group manipulation; and, message reception. In addition, the Session Manager is also responsible for spawning additional processes invoked by a client. These spawned processes can be placed anywhere within the network based on an application specific, or machine load dependent, algorithm.

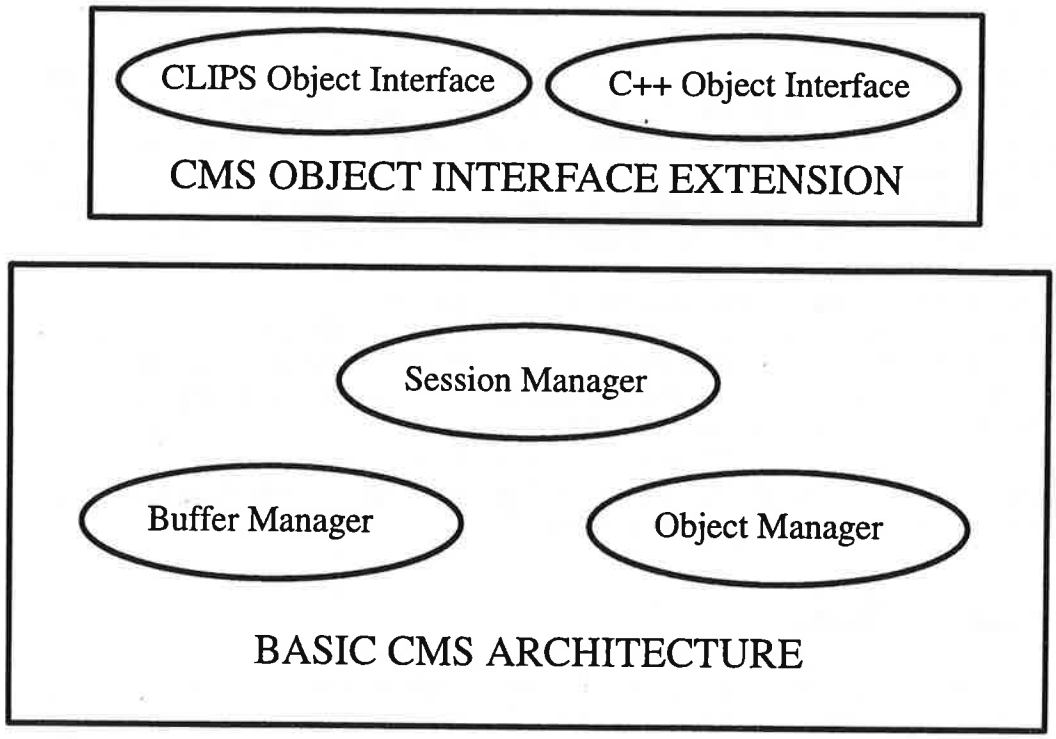


Figure 5.5: CMS architecture

5.5.2 Buffer Manager

As mentioned earlier, communication objects are placed into CMS buffers that can be passed back and forth between the various CMS clients. The Buffer Manager is responsible for providing the functionality required to create, fill, send, receive, parse, and finally destroy CMS buffers.

Clients are able to place any number of heterogeneous objects into a CMS buffer. Once filled, CMS buffers can be sent to other clients in either point-to-point or broadcast mode. When received by a target client, these CMS buffers can be queried and passed to

the preregistered client message handler. Clients can then parse these buffers by querying them as to their content. Each query reveals information about the next object in the buffer. In this manner, CMS clients are able to communicate groups of heterogeneous objects within the same CMS buffer.

5.5.3 Object Manager

The Object Manager is perhaps the most important component of the CMS adjunct. Responsible for representing all communicable objects within a CMS Session, the Object Manager provides its clients with the functionality required to encode any object into a CMS Slot List. These lists are designed to contain class and instance information in addition to a dynamic number of object attributes.

As mentioned earlier, CMS clients have the ability to send only those attributes of an object instance that are material to the communication. No additional attributes need to be sent for remote object creation and update. This is an important characteristic of CMS since communication costs among heterogeneous, distributed application environments can be high. However, an obvious disadvantage of this approach is the translation of a high level application object into a CMS Slot List. This translation is required primarily due to the fact that when dealing with heterogeneous networks, eventually all communicated data must be translated into the data type representation of the target platform. Lists provide a manageable medium to perform such translation. In addition, CMS Slot Lists provide a common representation for all application objects when passed between clients written in different languages.

5.6 Object Interface Architecture

CMS provides its more formal (predefined) object-oriented clientele with two additional interfaces: the CLIPS Interface; and, the C++ Interface. Both of these libraries reside at a level above the basic CMS system described in the previous Section. Many of the more cumbersome procedures required by clients interfacing with the basic CMS system, such as message handling and object translation, are implemented transparently by each of these interfaces, as described in the following two Sections.

5.6.1 The C++ Object Interface

The purpose of the C++ Object Interface is to provide object-based C++ clients with the ability to communicate C++ objects without requiring any form of translation (Stroustrup 1986). Further, received objects should be able to be processed in a completely transparent fashion by the receiver. In other words, remote object creation, update, and deletion must be performed in its entirety by the C++ Interface. A brief discussion of the procedures

employed by the C++ Interface to implement remote object manipulations, follows.

Both the C++ Interface and the CLIPS Interface utilize a common translation table to effectively provide mappings of CLIPS objects to C++ objects and vice versa. During compilation this table is used to generate a number of static lookup tables, which are used by the C++ Interface to map each object attribute, or slot, to an appropriate C++ method. These automatically generated tables form the core of the CMS Slot Lists to C++ objects translation mechanism.

The sending of a CMS object is a rather straightforward process. First, a CMS Slot List header is created representing the target object. Then an overloaded (i.e., single function name that deals with multiple classes) method called 'vAddSlot()' is invoked to add the individual slots to the CMS Slot List. After the CMS Slot List has been filled it is passed as an argument to any CMS 'send' or 'buffer' function. The selected CMS function determines whether the object is intended to be remotely created, updated, or deleted.

Once the remote object request is received by the destination client it is queried to ascertain whether the object is to be created, updated, or deleted. In the case of creation, the CMS Slot List's header is examined for the associated C++ class type. This value is used to look up a 'newobject' function for that class. This 'newobject' function is called with the CMS instance ID obtained from the object header. Once constructed, a reference to the new object is returned.

Similarly, when a remote update is requested, the class type is used to look up a 'getobject' function which takes a CMS instance ID and returns a reference to the existing object. Once obtained, the CMS dynamic attribute list contained in the CMS Slot List is traversed. For each slot the class type and slot type are used to look up a 'setmethod'. This 'setmethod' is then invoked to effectively set the correct value in the C++ object instance.

In addition, the class type and slot type are also used to look up a 'typecheck' character which represents the argument type that the 'setmethod' expects to receive. If this type differs from the actual type of the data contained in the current slot, a warning is flagged, but the operation is not interrupted. This process continues until the end of the attribute list is reached, thus completing the remote update.

Remote deletions are processed by using a 'getobject' function to obtain a reference to an existing object. Once this reference has been obtained, the C++ delete operator is called to invoke the appropriate class object destructor. If at any time an error occurs, a warning is produced and processing for that object or slot terminates.

5.6.2 The CLIPS Object Interface

The CLIPS Expert System Shell provides a rule-based programming language suitable for the development of knowledgebase systems (NASA 1989 and 1991). More recent versions of CLIPS offer developers an object-oriented sub-language known as CLIPS Object-Oriented Language (COOL). COOL allows users to represent and manipulate knowledge in terms of objects. The objective of providing a CMS interface to COOL was to facilitate the integration of rule-based application components into formal object-oriented systems.

Similar to the C++ Interface, the CLIPS Interface provides for the communication of high level application objects within a heterogeneous collection of CMS clients. In other words, it allows for the creation, update, and deletion of COOL objects from remote clients.

These operations are performed in a similar fashion to those of the C++ Object Interface. Every time a CMS buffer is received by a COOL client the buffer is parsed and each object processed. Processing an object involves determination of whether the required operation is a creation, update, or deletion. Once this information has been obtained the object instance ID is constructed, based on the information contained in the CMS Slot List header.

The creation of an object is accomplished by utilizing an instance name generated by encoding class and instance information located in the Slot List. Once the object instance has been obtained or constructed, the associated attribute list is processed by mapping slot identifiers to COOL slot names. Each object slot specified in the list is set with the corresponding value. In the case of remote deletions, the object instance is deleted directly from the CLIPS object list.

All of these operations take place in a transparent fashion to the associated client code. Once the communication transfer has been completed, control returns to the main CLIPS inference engine. At this point any rules depending on the modified objects are free to fire.

5.7 Current Status of CMS

The design of CMS was driven by the object-oriented nature of recent ICDM applications, with a particular focus on overall performance and data currency. This application focus is reflected in the mechanisms that were chosen to provide the desired functionality of the system. The principal elements of the design approach can be summarized in terms of: high level of abstraction; seamless interface among communicating components; support of distributed, heterogeneous network environments utilizing PVM; real-time event notification through preemption procedures; and, an

architecture that emphasizes atomicity.

The initial implementation of CMS described in this Section appears to validate this design approach in several respects. First, a high level object-based communication interface is presented to a heterogeneous collection of clients within a networked environment. Second, multi-processing efficiency and data currency are achieved through the utilization of interrupt-based preemption procedures. Third, application flexibility, extensibility, and simplicity are maintained as a result of the underlying theme of atomicity which is inherent within the overall design and implementation of CMS.

. CAD Research Center, Cal Poly, San Luis Obispo, Jun.'95 [CADRU-10-95]

6. References and Bibliography

Agha G.A.(1988); 'Actors'; MIT Press, Cambridge, Massachusetts.

ASP (1988); 'X Manual Set: X Intrinsics and Athena Widgets'; v.3, ASP Inc., San Jose, California.

Assal H. and L. Myers (1990); 'An Implementation of a Frame-Based Representation in CLIPS'; Proc. First CLIPS Users Conference, NASA, Lyndon B. Johnson Space Center, Houston, Texas, Aug.13-15 (570-580).

Bach M. (1986); 'The Design of the UNIX Operating System'; Prentice-Hall, Englewood Cliffs, New Jersey.

Beguelin A., J.Dongarra, G.A.Geist, R.Manchek and V.Sunderam (1991); 'A User's Guide to PVM: Parallel Virtual Machine'; Technical Report TM-11826, Oak Ridge National Laboratory, Oak Ridge, TN.

Berstein P.A., V. Hadzilacos, and N. Goodman (1987); 'Concurrency Control and Recovery in Database Systems'; Addison-Wesley, Reading, Massachusetts.

Birman K. and K. Marzullo (1989); 'ISIS and the META Project'; Sun Technology, Summer (pp.90-104).

Bomans L., D.Roose and R.Hempel (1990); 'The Argonne/GMD Marcos in FORTRAN for Portable Parallel Programming and their Implementation in the Intel iPSC/2'; Parallel Computing, 15 (119-132).

Booch G.(1991); 'Object Oriented Design'; Benjamin/Cummings, Redwood City, California.

Butler R. and E.Lusk (1992); 'User's Guide to the p4 Parallel Programming System'; Technical Report ANL-92/17, Argonne National Laboratory, October.

CADRC (1993); 'ICODES: Integrated Computerized Deployment System'; Final Report of Stage (1), Contract N47408-93-C-7347, CAD Research Center, Cal Poly, San Luis Obispo, CA.

Chaib-Draa B., R.Mandiau and P.Millot (1992); 'Distributed Artificial Intelligence: An Annotated Bibliography'; Sigart Bulletin, ACM Press, 3(3), August.

Conry S.E., R.A.Meyer and V.R.Lesser (1989); 'Multistage Negotiation in Distributed Planning'; in Bond and Gasser (eds.) Readings in Distributed Artificial Intelligence, Morgan Kaufmann, San Mateo, CA.

Davis R. and R.Smith (1983); 'Negotiation as Metaphor for

Distributed Problem Solving'; Artificial Intelligence, Vol.20, Jan. (pp.63-109).

Durfee E. and T.Montgomery (1990); 'A Hierarchical Protocol for Coordination of Multiagent Behavior'; Proc. 8th National Conference on Artificial Intelligence, Boston, Massachusetts (pp.86-93).

Durfee E.H.(1988); 'Coordination of Distributed Problem Solvers'; Kluwer Academic Publishers, London, England.

Durrant-Whyte H.F.(1988); 'Integration, Coordination and Control of Multi-Sensor Robot Systems'; Kluwer Acadenmic, Boston, Massachusettes.

Eicken (von) T., D.E.Culler, S.C.Goldstein and K.E.Schauser (1992); 'Active Messages: A Mechanism for Integrated Communication and Computation'; Proc. 19th Int. Symp. on Computer Architecture, Gold Coast, Australia, May (also available as Technical Report UCB/CSD 92/675, Computer Science Division, University of California at Berkeley, Berkeley, CA).

Elmasri R. and Navathe S.B.(1989); 'Fundamentals of Database Systems'; Benjamin/Cummings, Redwood City, California.

Erman L., F.Hayes-Roth, V.Lesser and D.Reddy (1980); 'The Hearsay-II Speech-Understanding System; Integrating Knowledge to Resolve Uncertainty'; Computing Surveys, 12(2).

Flower J., A.Kolawa and S.Bharadwaj (1991); 'The Express Way to Distributed Processing'; Supercomputing Review, May (54-55).

Forgy C. (1982); 'Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem'; Artificial Intelligence, 19 (17-37).

Geist G.A., A.Begguelin, J.Dongarra, W.Jiang, R.Manchek and V.Sunderam (1993); 'PVM-3 User's Guide and Reference Manual'; Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee, May.

Geist G.A., M.T.Heath, P.W.Peyton and P.H.Worley (1990); 'PICL: A Portable Instrumented Communications Library, C Reference Manual'; Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July.

Georgeff M. (1983); 'Communication and Interaction in Multi-Agent Planning'; Proc. Eighth Int. Joint Conf. on Artificial Intelligence, August (125-129).

Georgeff M. (1984); 'A Theory of Action for Multiagent Planning'; Proc. Fourth Nat. Conf. on Artificial Intelligence,

August (121-125).

Giarratano J. and G.Riley (1989); 'Expert Systems: Principles and Programming, PWS-Kent, Boston.

Gropp W.D., E.Lusk and A.Skjellum (1994); 'Using MPI: Portable Parallel Programming with the Message-Passing Interface'; MIT Press, Cambridge, Massachusettes.

Gropp W.D. and B.Smith (1993); 'Chameleon Parallel Programming Tools Users Manual'; Technical Report ANL-93/23, Argonne National Laboratory, March.

Halpern J.Y. and Y.Moses (1984); 'Knowledge and Common Knowledge in a Distributed Environment'; Proc. Third ACM Conf. on Principles of Distributed Computing.

Harrison R.J. (1991); 'Portable Tools and Applications for Parallel Computers'; International Journal Quantum Chemistry, 40(847).

Hayes-Roth B. (1985); 'A Blackboard Architecture for Control'; North-Holland: Artificial Intelligence, Elsevier Science Publishers.

Jones D. (1988); 'Introduction to the X-Window System'; Prentice-Hall, Englewood Cliffs, NJ.

Kerninghan B. and R.Pike (1984); 'The UNIX Programming Environment'; Prentice Hall, Englewood Cliffs, New Jersey.

Kerninghan B. and D.Ritchie (1988); 'The C Programming Language' 2nd ed., Prentice-Hall, Englewood Cliffs, NJ.

King P. and Collmeyer A.(1973); 'Database Sharing: an Efficient Mechanism for Supporting Concurrent processes'; in Proceedings of the NCC.

Krakowiak S. (1989); 'Principles of Operating Systems'; MIT Press, Cambridge, Massachusetts.

Kraus S. and J.Wilkenfeld (1990); 'The Function of Time in Cooperative Negotiations'; in Huhns (ed.) Proc. 10th International Workshop on Distributed Artificial Intelligence, Bandera, Texas.

Lansky A.L. (1985); 'Behavioral Specification and Planning for Multiagent Domains'; Technical Report 360, SRI International, Menlo Park, CA, November.

Larsi B., H.Larsi and V.Lesser (1990); 'Negotiation and its Role in Cooperative Distributed Problem Solving'; in Huhns (ed.) Proc. 10th Int. Workshop on Distributed Artificial Intelligence, Bandera, TX.

Leffler S., M.McKusick, M.Karels and J.Quarterman (1989); 'The Design and Implementation of the 4.3 BSD UNIX Operating System'; Addison Wesley, Reading, Massachusetts.

Mark Williams Company (1988); 'ANSI C: A Lexical Guide'; Prentice-Hall, Englewood Cliffs, New Jersey.

McGilton H. and R.Morgan (1983); 'Introducing the UNIX System'; McGraw-Hill, NY.

Miller L.H. and A.E. Quilici (1986); 'Programming in C'; Wiley and Sons, New York, New York.

MPI (1994); 'Message Passing Interface Forum: Mpi, a Message-Passing Interface Standard'; Technical Report CS-94-230, Computer Science Department, University of Tennessee.

Myers L. and J.Pohl (1994); 'ICDM: Integrated Cooperative Decision Making - in Practice'; 6th International Conference on Tools with Artificial Intelligence, New Orleans, Louisiana, November 6-9 (accepted).

NASA (1989); 'CLIPS Architecture Manual (Version 4.3)'; Artificial Intelligence Section, Lyndon B.Johnson Space Center, TX, May.

NASA (1991); 'CLIPS Reference Manual'; Artificial Intelligence Section, Johnson Space Center, Houston, TX. (distributed by COSMIC, University of Georgia, Athens, GA, USA).

Newell A. (1990); 'Unified Theories of Cognition'; Harvard University Press, Cambridge MA.

Pike R. et al. (1990); 'Plan 9 from Bell Labs'; Research Note, Bell Labs, July.

Pohl J. and L.Myers (1994); 'A Distributed Cooperative Model for Architectural Design'; Symposium on Knowledge-Based Systems for Architectural Design; Consiglio Nazionale Delle Ricerche, Rome, Italy, May 13-14, 1993: in Knowledge-Based Computer-Aided Architectural Design (eds. Carrara and Kalay), Elsevier, Amsterdam, (pp.205-242).

Pohl J., L.Myers and A.Chapman (1994); 'Thoughts on the Evolution of Computer-Assisted Design'; Technical Report, CADRU-09-94, CAD Research Center, Design and Construction Institute, College of Architecture and Environmental Design, Cal Poly, San Luis Obispo, CA, September.

Pohl J., J.LaPorta, K.Pohl and J.Snyder (1992); 'AEDOT Prototype(1.1): An Implementation of the ICADS Model'; Technical Report, CADRU-07-92, CAD Research Center, Design and Construction Institute, College of Architecture and Environmental Design, Cal Poly, San Luis Obispo, CA, August.

Pohl J., L.Myers, A.Chapman, J.Snyder, H.Chauvet, J.Cotton, C.Johnson and D.Johnson (1991); 'ICADS Working Model Version 2 and Future Directions'; Technical Report CADRU-05-91, CAD Research Unit, Design Institute, Cal Poly, San Luis Obispo, California.

Pohl J., L.Myers and A.Chapman (1990); 'ICADS: An Intelligent Computer-Aided Design Environment'; St. Louis: Proc. ASHRAE Conference, June 9-13.

Pohl J., L.Myers, A.Chapman and J.Cotton (1989); 'ICADS: Working Model Version 1'; Technical Report, CADRU-03-89, CAD Research Unit, Design Institute, School of Architecture and Environmental Design, Cal Poly, San Luis Obispo, CA.

Pohl J., A.Chapman, L.Chirica, R.Howell and L.Myers (1988); 'Implementation Strategies for a Prototype ICADS Working Model'; Technical Report, CADRU-02-88, CAD Research Unit, Design Institute, School of Architecture and Environmental Design, Cal Poly, San Luis Obispo, CA.

Pohl K.J. (1992); 'GXI: A Graphical Interface Builder with Distributed Inter-Process Message Management Capabilities'; Focus Symposium on Computer-Based Design Environments, 6th International Conference on Systems Research, Informatics and Cybernetics, Baden-Baden, Germany, Aug.18-19.

Pohl K.J. (1991); 'A CLIPS/X-WindowInterface'; 2nd CLIPS Users Conference Proceedings, Houston, Texas, September 23-25.
Renzetti R. (1995); 'Architectural Design Program Loader for ICADS-DEMO2'; Master of Science in Architecture Thesis, Architecture Department, Cal Poly, San Luis Obispo, CA.

Renzetti R. (1995); 'Architectural Design Program Loader for ICADS-DEMO2'; Master of Science in Architecture Thesis, Architecture Department, Cal Poly, San Luis Obispo, CA.

Rosenberry W., D.Kenney and G Fisher (1992); 'Understanding DCE'; O'Reilly, Sebastopol, CA.

Rosenshein J.S. and M.R.Genesereth (1987); 'Communication and Cooperation Among Logic-Based Agents'; Proc. Sixth Phoenix Conf. on Computers and Communications, February (594-600).

Rosenshein J.S. and M.R.Genesereth (1985); 'Deals Among Rational Agents'; Proc. Ninth Int. Joint Conf. on Artificial Intelligence, August (91-99).

Sunderam V. (1990); 'PVM: A Framework for Parallel Distributed Computing'; Concurrency: Practice & Experience, 2(4), Dec.

Skjellum A., S.G.Smith, C.H.Still, A.P.Leung and M.Morari (1994); 'The Zipcode Message-Passing System'; in G.Fox (ed.) Parallel Computing Works!, Morgan Kaufmann, San Mateo, CA.

Smith J.B. (1994); 'Collective Intelligence in Computer-Based Collaboration'; Lawrence Erlbaum, Hillsdale, NJ.

Smith R.G. (1980); 'The Contract-Net Protocol: High-Level Communication and Control in a Distributed Problem Solver'; IEEE Transactions on Computers, C-29(12), Dec. (pp.1104-13).

Stevens W.R. (1990); 'UNIX Network Programming'; Prentice-Hall, Englewood Cliffs, New Jersey.

Stroustrup B. (1986); 'The C++ Programming Language'; Addison-Wesley, Reading MA.

Sycara K. (1989); 'Multiagent Compromise via Negotiation'; in Gasser and Huhn (eds.) Distributed Artificial Intelligence, Vol.II, Pitman, Los Altos, California (pp.119-138).

Tanenbaum A.S. (1987); 'Operating Systems: Design and Implementation'; Prentice-Hall, Englewood Cliffs, NJ.

Tanenbaum A.S. and R.V.Renesse (1985); 'Distributed Operating Systems', ACM Computing Surveys. 17(4), December.

Taylor J (1990); 'A Framework for Multiple Cooperating Agents in an Intelligent CAD Environment'; Master of Science in Architecture Thesis, Architecture Department, Cal Poly, San Luis Obispo, CA.

Taylor J. and L.Myers (1990); 'Executing CLIPS Expert Systems in a Distributed Environment'; Proc. First CLIPS Users Conference, NASA, Lyndon B. Johnson Space Center, Houston, Texas, Aug.13-15 (686-695).

Taylor J. and J.Pohl (1990); 'A Geometry Interpreter for Extracting Architectural Objects from the Point/Line Schema of a CAD Database'; Proc. InterSymp-90, Baden-Baden, Germany, August 6-12.

Turcotte L. (1993); 'A Survey of Software Environments for Exploiting Networked Computing Resources'; Draft Report, Mississippi State University, Jackson, Mississippi, January.

Williams (1988), Mark Williams Company; 'ANSI C: A Lexical Guide'; Prentice-Hall, Englewood Cliffs, NJ.

Young D. (1990); 'The X Window System: Programming and Applications with Xt (OSF/Motif Edition)'; Prentice-Hall, Englewood Cliffs, NJ.

7. Appendix A: Interfacing ICADS Agents

The first version of the ICADS prototype (referred to as DEMO1 in this Appendix) was designed to facilitate the interface of new agents (referred to as IDTs (Intelligent Design Tools) in this Appendix) into the system. Although the CLIPS expert system shell was used to write most of the IDTs, all uses of CLIPS in communication with the IDTs are duplicated by alternative functions written in the C language. This makes it possible to choose between CLIPS and C code when adding new IDTs, or other software components, that will use the IDT interface. It is also possible to mix levels of CLIPS and C functions to provide a variety of software interfacing combinations in DEMO1.

The essential interface ability is communication with the blackboard. In the DEMO1 architecture the blackboard is actually a central control facility, referred to as the Blackboard Control System, that reacts to the information generated by the domain experts (i.e., IDTs). It functions primarily as a CLIPS expert system that distributes the facts that describe the essential state of the design project. Since the IDTs are written in CLIPS the interface between the Blackboard Control System and the IDTs is oriented to the manner in which CLIPS interacts with external functions. However, it is possible to link virtually any software component via the IDT interface, by replacing the CLIPS specific code with equivalent C functions. By following the conventions established specifically for the IDTs written in CLIPS, any other process can communicate with the Blackboard Control System and hence to all other software components connected to the Blackboard.

A summary description of the structure of a typical IDT written in CLIPS will provide the basis for understanding how any component can interface with the system. Portions of the 'Cost' IDT, an agent which represents the construction cost domain in the ICADS prototype, will be used to demonstrate the essential features.

UNIX sockets provide the mechanism for connecting the IDTs to the Blackboard Control System. However, it is not necessary to be directly involved with any of the socket software in order to interface with the system. All of the software needed is provided by C functions that can be called as user functions in CLIPS, or as simple C functions from any other environment.

The Blackboard Knowledgebase

The blackboard knowledgebase is the file 'bb.kb'. It must know of each IDT that is to interface with the system. When DEMO1 is executed this CLIPS ruleset initiates the execution of each of the IDT processes, via a function called 'system', with the name of the IDT's start file. Each IDT requires three files, identified by extensions '.start', '.load' and '.kb':

```
start file ..... (e.g., 'cost.start')
load file ..... (e.g., 'cost.load')
knowledgebase file .... (e.g., 'cost.kb')
```

The start file is a shellscript file that provides for automatic generation of the process on the proper host machine with X-Window output from the process to the designated host display. The load file contains a set of CLIPS commands that will load the IDT ruleset and the CLIPS rules from a file called 'bbint.kb', before commencing execution of the CLIPS IDT rules. The rules in the latter file are responsible for receiving facts from the Blackboard Control System and performing the actions they identify on the local factlist. All load files for CLIPS IDTs are similar. The 'cost.start' and 'cost.load' files are typical of all start and load files. When a non-CLIPS component is added to DEMO1 a few lines of code must be added to initiate the new process.

In addition, it is necessary to duplicate in the new IDT software the effect of some, or all, of the rules located in the 'bbint.kb' file. Four rules handle the reception of facts from the blackboard and the rest perform local actions on the IDT factlist. Generally, these facts tell the IDT to add, modify, or delete a fact from its own factlist in order to agree with the information on the blackboard. It is quite easy to duplicate these rules in another knowledgebase language, as long as the latter permits the execution of external functions.

From CLIPS to C

When a new IDT is added the changes necessary to invoke the new IDT should be the responsibility of the systems programmer. Therefore, this section is restricted to describing the responsibilities of the IDT developer.

First the IDT must help to establish the socket communication with the blackboard. For example, one of the first rules in the 'Cost' IDT accomplishes this requirement as follows:

```
(defrule  begin-input-template
          ?initial <-  (initial-fact)
=>
          (new_server "cost")
          (assert (BBASSERT =(connect_bb "bb")))
          (retract ?initial)
)
```

The CLIPS user-functions, 'new_server' and 'connect_bb', are used in the above CLIPS rule to call the C functions 'u_new_server' and 'u_connect_bb', respectively. As with all external functions to CLIPS, this equivalence is accomplished through a declaration in the CLIPS 'usrfuncs' function which is found in the CLIPS 'main.c' file. Further, all that is needed to add the 'new_server' function to CLIPS is an external declaration and a statement of the form:

```
define_function ("new_server",'i',u_new_server,"u_new_server");
```

This 'define_function' provides the name of the CLIPS function, the type of value which it returns (i.e., 'i' is for 'int') and the name of the actual function. The last argument is always the string form of the third argument. When the CLIPS system is recompiled, the external function will be linked into all calls made from the CLIPS rules.

The CLIPS external functions are never defined with arguments, even though they may be called with arguments. Instead of defining arguments, the following special functions are called within the user-defined functions to access the actual arguments:

int	num_args()	returns the number of arguments
char	*rstring(arg)	returns a pointer to the 'arg' argument, assumed to be a string
float	rfloat(arg)	returns the floating point number, assumed to be the 'arg' argument
int	runknown(arg)	returns an int that identifies the type of argument for 'arg'

For example, inside 'u_new_server' a statement such as 'n = num_args();' will set 'n' to the number of arguments with which the function was called. The actual argument can be retrieved by calling 'rstring(1)', which returns a pointer to the character string. If the argument is a floating point number, 'rfloat(1)' will return the floating point value. Further, if the type of argument is not known, 'runknown(1)' will return an integer that identifies the type of argument. It should be noted that the value '1' is used in these examples to denote the first argument. The number of the argument must be used for each argument value one wishes to access. Specifically in the above rule from the 'Cost' IDT, a call to 'num_args()' would return '1' and the argument 'cost' is retrieved in 'u_new_server' by calling 'rstring(1)'.

For non-CLIPS IDTs, a set of C functions that duplicate the effect of the CLIPS external functions is used. These C functions use standard arguments and hence do not rely on the special argument transmission necessary for CLIPS IDTs.

For example, if the C function 'new_server("cost")' is executed, exactly the same result will occur as if the call to 'new_server' from within the CLIPS IDT had been made. Similarly, if the C function 'connect_bb("bb")' is called, exactly the same connection to the Blackboard will be made as in the rule from the CLIPS 'Cost' IDT above.

Thus it is possible to develop an IDT in any system that permits external C functions to be called. Every function developed for IDTs written in CLIPS will continue to be duplicated by a C

function that uses standard argument transfer!

Declaring an IDT

An IDT, or any process wishing to use the IDT interface, must be declared in the 'icads.conf' system configuration file. Each IDT is given a unique identification number, called its key value, to guarantee system integrity. The entry consists of one line that gives the name of the IDT, its key value, the machine on which the process should run, and the display unit that should receive its X-Window display. For example, the entry for the 'Cost' IDT might be as follows:

```
cost            403     jellyfish       angelfish:0
```

(i.e., for IDT 'Cost' the identification number is '403', it will execute on the computer with the symbolic network name 'jellyfish', and display on the monitor of computer 'angelfish' in X-Window session '0')

In the example above, the 'u_new_server' function checks the 'icads.conf' file for the name of the IDT that it has been given as an argument. Similarly, the 'u_connect_bb' function makes a call to the 'new_client' function which also checks the 'icads.conf' file to validate the IDT.

Specifying the Blackboard Values

After the blackboard connection is established, by the 'connect_bb' call, the CLIPS IDT asserts a template of facts to the blackboard factlist. These fact templates, with the value 'input' as the third field, identify the form of all facts that the IDT wishes to receive. Thus, the Blackboard Control System only sends to the IDT those facts in which it is interested. This reduces the communication costs and keeps the CLIPS network from being slowed down by the transfer of unnecessary facts.

For example, in the rule from the 'Cost' IDT discussed previously the connection to the blackboard results in the assertion of a fact of the form '(BBASSERT nn)', where 'nn' is the identifier of the IDT and the socket interface. The succeeding rule which is executed next, has the following form:

```
(defrule input-template
        (BBASSERT ?no)
=>
        (bb_assert (ADD FRAME idt ?no)
             (ADD VALUE idt input ?no FRAME space)
             (ADD VALUE idt input ?no FRAME door)
             (ADD VALUE idt input ?no FRAME window)
             (ADD VALUE idt input ?no VALUE space area)
             (ADD VALUE idt input ?no VALUE space name)
```

```
                    (ADD VALUE idt input ?no VALUE space perimeter)
                    (ADD VALUE idt input ?no RELATION space wall)
                    (ADD VALUE idt input ?no RELATION space light)
                    (ADD VALUE idt input ?no RELATION space column)
                    (ADD VALUE idt input ?no RELATION space table)
                    (ADD VALUE idt input ?no RELATION space chair)
                    (ADD VALUE idt input ?no RELATION space door)
                    (ADD VALUE idt input ?no RELATION space window)
                    (ADD VALUE idt input ?no VALUE door width)
                    (ADD VALUE idt input ?no VALUE window width)
                    (ADD VALUE idt violation-char ?no "C")
                )
        )
```

The first field of any fact in a 'bb_assert' will either be 'ADD', 'MODIFY', or 'DELETE'. When the first field is 'ADD', the Blackboard Control System will add a fact consisting of the rest of the fields to the blackboard factlist. What then happens with the fact depends on the third and fourth field values. The facts with 'input' are not transmitted to the IDTs that are connected to the blackboard. Instead, the Blackboard Control System uses them as templates to determine which facts are sent to the designated IDT. In other words, when facts that match the pattern of any of the template facts are later asserted to the blackboard, those facts will be sent to the IDT identified by the field following 'input'.

The function 'bb_assert' is not a CLIPS external function. In order to be able to use the same syntax for 'bb_assert' as that used in the normal CLIPS 'assert' statement, the 'bb_assert' function was inserted into the CLIPS source code. However, for other environments C functions have been written that duplicate the effect of 'bb_assert'. These can be called from a non-CLIPS environment to transmit equivalent fact information to the blackboard.

One additional factor is important for understanding how the facts are sent through the sockets. It is frequently desirable to have a set of facts communicated as a group, during the execution of CLIPS rules. Therefore, the functions that handle the sockets are written to group facts as a single message until an 'end of message' signal is received. In other words, each message sent via the socket is a series of facts. The number of facts per message is variable and under the complete control of the sender.

In the 'Cost' IDT the following rule will fire after the one shown above, so that all of the facts will be received as a single event:

```
        (defrule end-input-template
                (declare (salience -100))
                (BBASSERT ?no)
        =>
                (bb_end_message)
        )
```

The user function 'bb_end_message' calls the C function 'u_bb_end_message' to send the appropriate message termination; and, of course, an equivalent 'bb_end_message()' C function exists.

Sending Facts to the Blackboard

The previous section described the same functions that are used to send facts from an IDT to the Blackboard. Typically a rule, or group of rules, will execute a 'bb_assert':

```
(bb_assert (MODIFY VALUE space result-cost ?space ?value))
(bb_assert (MODIFY VALUE idt violation-on ?no ?space))
```

Then, after all of the assertions that should be grouped together have been made, an execution of 'bb_end_message' follows.

Receiving Facts from the Blackboard

The process of receiving facts is more complex than sending facts. This is partially due to the expectation that actions should take place within the IDT as a result of receiving a fact. For example, if a fact incorporating a DELETE action is received for a VALUE slot of a frame, then this fact should be deleted from the IDT factlist. In a CLIPS IDT this particular case is handled by a rule from the 'bbint.kb' file, which is of the following form:

```
(defrule  delete-value
        (declare (salience -30)
        ?con <- (DELETE VALUE ?class ?field ?frame $?values)
        ?old <- (VALUE ?class ?field ?frame $?values)
=>
        (retract  ?con  ?old)
)
```

It should be noted that this is a generic rule which will handle all instances for deletions, of this form. Several similar rules are needed to enable a CLIPS IDT to respond to all actions that are necessary to update the IDT factlist.

Also, there is the question of when the socket should be checked for reception of a fact. In the CLIPS IDTs a low priority loop consisting of four rules is used for this purpose. Whenever the active processing of facts is completed, the socket are checked, and all facts that have been sent as a grouped message are asserted into the local factlist. The user function 'receive_message' which calls the C function 'u_receive_message' performs this function.

Creating an IDT not Written in CLIPS

The DEMO1 code that interfaces an IDT with the blackboard is based on the premise that the information transmitted is a hashed form of

a CLIPS fact. An interface with non-CLIPS software is most easily accomplished by duplicating the effect of the functions that deal with CLIPS facts, as high in the communication hierarchy as possible.

The communication hierarchy begins at the lowest level with the message passing C functions, the socket handling routines found in the 'message.c' file. At a level above these functions are the routines resident in the 'factio.c' file. These routines call the message passing functions from a viewpoint that corresponds to the way facts are seen in the CLIPS world. Finally, two parallel sets of functions in the files 'kbio.c' and 'bbio.c' are used to communicate with the 'fact' functions from a CLIPS ruleset or from a C environment, respectively.

If another expert system shell (i.e., other than CLIPS) is used for an IDT, then the recommended method for interfacing with the DEMO1 system is given below:

1. Rewrite, in the given shell language, the CLIPS rules found in the 'bbint.kb' file using the C functions from the 'bbio' viewpoint to duplicate the effect of 'receive_message'.

2. Write rules similar to the 'begin-input-template', 'input-template' and 'end-input-template' examples from a CLIPS IDT, such as those in 'cost.kb', using the 'bbio' C functions for 'new_server', 'connect_bb' and 'bb_end_message'.

3. Modify, as appropriate, the code that initiates the execution of the new IDT.

4. Write rules in the given shell language, with calls to the generic C functions to assert values to the blackboard using the 'bbio' C functions to duplicate the CLIPS 'bb_assert' statement.

If a software environment other than an expert system is to interact with the Blackboard that interaction should also take place through the C functions from the 'bbio' module. However, in this case the CLIPS rules in the 'bbint.kb' file may not provide much help to determine what should happen in response to the actions described in the facts received from the blackboard. The response must be determined in relation to the reasons for the addition of the new software component.

Example: File 'cost.kb' for 'Cost' IDT

```
;*****************************************************************
;
;       ICADS-DEMO1:    'cost.kb' for COST IDT                (5/20/90)
;
;*****************************************************************
```

```
(deffacts prices
        (price wall sqft 10)
        (price door sqft 4)
        (price floor sqft 50)
        (price light unit 80)
        (price column unit 300)
        (price chair unit 100)
        (price table unit 200)
        (maxcost RESTROOM 12000)
        (maxcost LOBBY 28000)
        (maxcost MANAGER 14400)
        (maxcost OFFICE-1 11200)
        (maxcost OFFICE-2 11200)
        (maxcost OFFICE-3 11200)
        (maxcost WAITING 16000)
        (maxcost STAFFROOM 16000)
        (maxcost LIBRARY-AUDITORIUM 80000)
        (maxcost STORE 16000)
        (maxcost CONFERENCE 12000)
        (maxcost DEFAULT 18000)
)


(defrule begin-input-template
        ?initial <- (initial-fact)
  =>
        (new_server "cost")
        (assert (BBASSERT =(connect_bb "bb")))
        (retract ?initial)
)



(defrule input-template
        (BBASSERT ?no)
  =>
        (bb_assert (ADD FRAME idt ?no)
                (ADD VALUE idt name ?no cost)
              (ADD VALUE idt input ?no FRAME space)
                 (ADD VALUE idt input ?no FRAME wall)
                 (ADD VALUE idt input ?no FRAME door)
                 (ADD VALUE idt input ?no FRAME window)
              (ADD VALUE idt input ?no VALUE space area)
              (ADD VALUE idt input ?no VALUE space name)
              (ADD VALUE idt input ?no VALUE space perimeter)
              (ADD VALUE idt input ?no RELATION space wall)
              (ADD VALUE idt input ?no RELATION space light)
              (ADD VALUE idt input ?no RELATION space column)
              (ADD VALUE idt input ?no RELATION space table)
              (ADD VALUE idt input ?no RELATION space chair)
              (ADD VALUE idt input ?no RELATION wall door)
              (ADD VALUE idt input ?no RELATION wall window)
              (ADD VALUE idt input ?no VALUE door width)
              (ADD VALUE idt input ?no VALUE window width)
```

```
                (ADD VALUE idt violation-char ?no "C")
    )
)

(defrule end-input-template
        (declare (salience -100))
        (BBASSERT ?no)
  =>
        (bb_end_message)
)

(defrule add-floor-cost
        ?mod <- (ADD VALUE space area ?space ?area)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price floor sqft ?x)
  =>
        (retract ?cost ?mod)
        (assert (VALUE space area ?space ?area))
        (assert (VALUE space cost ?space =(+ ?total-cost (* ?area ?x))))
)

(defrule modify-floor-cost
        ?mod <- (MODIFY VALUE space area ?space ?area)
        ?oldval <- (VALUE space area ?space ?oldarea)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price floor sqft ?x)
  =>
        (retract ?cost ?mod ?oldval)
        (assert
         (VALUE space area ?space ?area)
         (VALUE space cost ?space =(- (+ ?total-cost (* ?area ?x))(* ?olda
         )
)

(defrule add-wall-cost
        ?mod <- (ADD VALUE space perimeter ?space ?perimeter)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price wall sqft ?x)
  =>
        (retract ?cost ?mod)
        (assert (VALUE space cost ?space =(+ ?total-cost (* ?perimeter 8 ?
                (VALUE space perimeter ?space ?perimeter))
)

(defrule modify-wall-cost
        ?mod <- (MODIFY VALUE space perimeter ?space ?perimeter)
        ?oldval <- (VALUE space perimeter ?space ?oldperimeter)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price wall sqft ?x)
  =>
        (retract ?cost ?mod ?oldval)
        (assert (VALUE space perimeter ?space ?perimeter))
        (assert (VALUE space cost ?space =(- (+ ?total-cost (* ?perimeter
)
```

```
(defrule add-window-cost
        ?mod <- (ADD VALUE window width ?window ?width)
        (RELATION wall window ?wall ?window)
        (RELATION space wall ?space ?wall)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price window sqft ?x)
  =>
        (retract ?cost ?mod)
        (assert (VALUE space cost ?space =(+ ?total-cost (* ?width ?x))))
        (assert (VALUE window width ?window ?width))
 )

(defrule modify-window-cost
        ?mod <-(MODIFY VALUE window width ?window ?width)
        ?oldval <-(VALUE window width ?window ?oldwidth)
        (RELATION wall window ?wall ?window)
        (RELATION space wall ?space ?wall)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price window sqft ?x)
  =>
        (retract ?cost ?mod ?oldval)
        (assert (VALUE window width ?window ?width))
        (assert (VALUE space cost ?space =(- (+ ?total-cost
                (* ?width ?x)) (* ?oldwidth ?x))))
 )

(defrule delete-window-cost
        (not (DELETE FRAME space ?space))
        ?mod <- (DELETE VALUE window width ?window ?width)
        ?oldval <- (VALUE window width ?window ?width)
        (RELATION wall window ?wall ?window)
        (RELATION space wall ?space ?wall)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price window sqft ?x)
  =>
        (retract ?cost ?mod ?oldval)
        (assert (VALUE space cost ?space =(- ?total-cost (* ?width ?x))))
 )

(defrule add-door-cost
        ?mod <- (ADD VALUE door width ?door ?width)
        (RELATION wall door ?wall ?door)
        (RELATION space wall ?space ?wall)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price door sqft ?x)
  =>
        (retract ?cost ?mod)
        (assert (VALUE space cost ?space =(+ ?total-cost (* ?width ?x))))
        (assert (VALUE door width ?door ?width))
 )

(defrule modify-door-cost
        ?mod <-(MODIFY VALUE door width ?door ?width)
        ?oldval <-(VALUE door width ?door ?oldwidth)
```

```
        (RELATION wall door ?wall ?door)
        (RELATION space wall ?space ?wall)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price door sqft ?x)
  =>
        (retract ?cost ?mod ?oldval)
        (assert (VALUE door width ?door ?width))
        (assert (VALUE space cost ?space =(- (+ ?total-cost
                (* ?width ?x)) (* ?oldwidth ?x))))
 )

(defrule delete-door-cost
        ?mod <- (DELETE VALUE door width ?door ?width)
        ?oldval <- (VALUE door width ?door ?width)
        (RELATION wall door ?wall ?door)
        (RELATION space wall ?space ?wall)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price door sqft ?x)
        (not (DELETE FRAME space ?space))
  =>
        (retract ?cost ?mod ?oldval)
        (assert (VALUE space cost ?space =(- ?total-cost (* ?width ?x))))
 )

(defrule add-light-cost
        ?mod <- (ADD RELATION space light ?space ?light)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price light unit ?x)
  =>
        (retract ?cost ?mod)
        (assert (VALUE space cost ?space =(+ ?total-cost ?x)))
        (assert (RELATION space light ?space ?light))
 )

(defrule delete-light-cost
        ?mod <- (DELETE RELATION space light ?space ?light)
        ?oldrel <- (RELATION space light ?space ?light)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price light unit ?x)
        (not (DELETE FRAME space ?space))
  =>
        (retract ?cost ?mod ?oldrel)
        (assert (VALUE space cost ?space =(- ?total-cost ?x)))
 )

(defrule add-column-cost
        ?mod <- (ADD RELATION space column ?space ?column)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price column unit ?x)
  =>
        (retract ?cost ?mod)
        (assert (VALUE space cost ?space =(+ ?total-cost ?x)))
        (assert (RELATION space column ?space ?column))
 )
```

```
(defrule delete-column-cost
        ?mod <- (DELETE RELATION space column ?space ?column)
        ?oldrel <- (RELATION space column ?space ?column)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price column unit ?x)
        (not (DELETE FRAME space ?space))
  =>
        (retract ?cost ?mod ?oldrel)
        (assert (VALUE space cost ?space =(- ?total-cost ?x)))
 )

(defrule add-chair-cost
        ?mod <- (ADD RELATION space chair ?space ?chair)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price chair unit ?x)
  =>
        (retract ?cost ?mod)
        (assert (VALUE space cost ?space =(+ ?total-cost ?x)))
        (assert (RELATION space chair ?space ?chair))
 )

(defrule delete-chair-cost
        ?mod <- (DELETE RELATION space chair ?space ?chair)
        ?oldrel <- (RELATION space chair ?space ?chair)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price chair unit ?x)
        (not (DELETE FRAME space ?space))
  =>
        (retract ?cost ?mod ?oldrel)
        (assert (VALUE space cost ?space =(- ?total-cost ?x)))
 )

(defrule add-table-cost
        ?mod <- (ADD RELATION space table ?space ?table)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price table unit ?x)
  =>
        (retract ?cost ?mod)
        (assert (VALUE space cost ?space =(+ ?total-cost ?x)))
        (assert (RELATION space table ?space ?table))
 )

(defrule delete-table-cost
        ?mod <- (DELETE RELATION space table ?space ?table)
        ?oldrel <- (RELATION space table ?space ?table)
        ?cost <- (VALUE space cost ?space ?total-cost)
        (price table unit ?x)
        (not (DELETE FRAME space ?space))
  =>
        (retract ?cost ?mod ?oldrel)
        (assert (VALUE space cost ?space =(- ?total-cost ?x)))
 )

(defrule initialize-cost
```

```
        (ADD FRAME space ?space)
        (not (VALUE space cost ?space ?value))
  =>
        (assert (FRAME space ?space))
        (assert (VALUE space cost ?space 0))
 )

(defrule finish-rule
        (declare (salience -20))
        (BBASSERT ?no)
        (VALUE space cost ?space ?total-cost)
        (VALUE space name ?space ?name)
        (or (maxcost ?name ?max-cost)
                (and (maxcost DEFAULT ?max-cost)
                                (not (maxcost ?name ?))))
  =>
        (bind ?value (- ?max-cost ?total-cost))
        (bb_assert (MODIFY VALUE space result-cost ?space ?value))
        (if (< ?value 0) then
                (bb_assert (MODIFY VALUE idt violation-on ?no ?space))
         else
                (bb_assert (MODIFY VALUE idt violation-off ?no ?space))
        )
        (assert (send))
 )

(defrule send-rule
                ?f <- (send)
        =>
                (retract ?f)
                (bb_end_message)
)
```

Example: Use of 'bbio' Functions

```
/*
 Connect -- connection module for controller of UI to blackboard

  TYPES:

    UNSIGNED_ CHAR_ BYTE_              A byte value
    UNSIGNED_ INT_  ID_                A generic ID
    BYTE_           *PTR_              A generic pointer
    ID_             HAND_ID_           A data handler ID
*/

#include <stdio.h>
#include "util/types.h"
#include "util/bbio.h"

#define IDTCLASS "idt"
void sig_handle();
```

```
typedef struct {
  INT_ numWords;
  CHAR_ *words[20];
} WORDLIST_;

STATIC_ INT_ myID;
STATIC_ WORDLIST_ words;

STATIC_ VOID_ read_words();
void print_error();

/*VARARGS*/
void print_error (a1, a2, a3, a4, a5, a6, a7, a8, a9, a10)
  char *a1;
  unsigned int a2, a3, a4, a5, a6, a7, a8, a9, a10;
  {
    fprintf (stderr, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10);
    fflush (stderr);
  }

INT_    dmInit()
{
  FILE *f;
  myID = connect_bb("bb", "ui");

/*
  Send list of frames I need to message handler
  (read from the file 'ui.frames')
*/

  f = fopen("ui.frames","r");
  if (f==NULL)
     return(1);

  write_bbframe (ADDACTION, IDTCLASS, myID);
  read_words(f,&words);

  while (!feof(f))
  {
    if ((!strncmp(words.words[0], "FRAME"))&&(words.numWords==2))
      write_bbwordvalues (ADDACTION, IDTCLASS, "input", myID,
                FRAMEFACT, words.words[1], NULL);
    if ((!strncmp(words.words[0], "VALUE"))&&(words.numWords==3))
      write_bbwordvalues (ADDACTION, IDTCLASS, "input", myID,
                VALUEFACT, words.words[1], words.words[2], NULL);
    if ((!strncmp(words.words[0], "RELATION"))&&(words.numWords==3))
      write_bbwordvalues (ADDACTION, IDTCLASS, "input", myID,
                RELATIONFACT, words.words[1], words.words[2], NULL);
    read_words(f, &words);
  }

  fclose(f);

  write_bbheader (SENTINAL);
```

```
   send_bbmessage ();
   build_bbhtable ();

         signal_bbmessage(sig_handle);
         return(0);
}

char * parser_buffer;

void sig_handle()
{
         INT_ numElements;
         INT_ count;
         CHAR_ * action;
         FLOAT_ * fp;
         INT_ type;

         parser_buffer = (char *) malloc(256);

         action = (char *) malloc(256);
         fp = (float *) action;
         receive_bbmessage();
         while( (numElements = read_bbheader()) != SENTINAL) {
                  type = read_bbelement(action);
                  numElements--;
                  fprintf(stderr,"            %s ", action);
                  type = read_bbelement(action);
                  numElements--;
                  fprintf(stderr,"    2nd action  %s ", action);
                           fflush(stderr);
                  for (count = 0 ; count < numElements ; count++) {
                           type = read_bbelement(action);
                           switch (type) {
                           case NUMBER:
                                    fprintf(stderr,"Case Number %f ", *fp);
                                    break;
                           case WORD:
                           case STRING:
                                    fprintf(stderr,"Case word or String %s ",
                                    action);
                                    break;
                           }
                  }
/*  Call to routines yyparse(), or for testing print_buffer()  */
         }
         free(action);
}

STATIC_ VOID_   read_words(f, words)
FILE *f;
WORDLIST_ *words;
{
  STATIC_ CHAR_ line[256];
  CHAR_ *p;
```

```
  INT_  c;

  do {
    p=line;
    while (((c=getc(f))!=EOF)&&(c!='\n'))
     *(p++)=(char)c;
    *p='\0';
  } while (line[0] == '#');

  p=line;
  c=0;
  while (*p) {
    words->words[c++]=p;
    while ((*p)&&(*p != ' '))
      p++;
    if (*p == ' ') {
      *(p++) = '\0';
      while (*p == ' ')
        p++;
    }
  }
  words->numWords=c;
}

main()
{
        dmInit();
        while (1);
}
```

8. Appendix B: GXI User Guide and Message Structure

The following is an example of an interactive session between a client and the GXI server. In this example the client creates a simple user interface in the GXI environment.

```
(deffacts
      buttons (Buttons "Access Database" "File System" "Help"
      "Exit")
)

(defrule CreateThermalInterface
      (Buttons $?MenuButtons)
=>
      ; *********  Request a Connection to GXI *****************
      (bind ?Sheet  ( XCInit?Client ?ColorFile ?XPos ?YPos
            ?Height ?Width ?BorderWidth ?BorderColor
            ?BkgColor )
      )

      ; *********  Create an Interface Banner  *****************
      (bind ?Banner ( XCBanner ?Sheet ?BannerText ?BorderWidth
            ?Length ?TextColor ?BkgColor ?FontStyle )
      )

      ; *********  Create the Main Menu  ***********************
      (bind ?Menu( XCMenu?Sheet NULL ?Banner ?BorderWidth
            ?BorderColor ?TextColor ?BkgColor ?Horizontal
            ?NumButtons $?MenuButtons )
      )

      ; *********  Assert the New Environment  *****************
      (assert(ThermalSheet  ?Sheet))
      (assert(ThermalBanner ?Banner))
      (assert(ThermalMenu?Menu))
)
```

Utilizing the pattern matching capabilities in CLIPS, the following typical rule could be associated with each menu button to react to user actions:

```
(defrule AccessDatabase
      ; *********  Match on the Database Button *************
      ?Selection <- (Thermal "Access Database")
=>
      ; *********  Perform Thermal Database Access **********
      ;  ...
      ;  ...
      (retract ?Selection)
)
```

A full set of GXI commands is available to clients written in either the 'C' language or CLIPS, as follows:

```
/*****************************************************************
PUBLIC Function Declarations
*****************************************************************/

PUBLIC int
XCInit ();
/*****************************************************************
* FUNCTION: Initializes the interface connection to the server   *
*             Creates the initial working sheet.                 *
* RETURNS:  WID (integer) of the SHEET                           *
*****************************************************************/
/*              client_name,    client name in icads.conf       */
/*              color_file,     file that holds color names     */
/*              x,              upper left hand x position      */
/*              y,              upper left hand y position      */
/*              height,         form height in pixels           */
/*              width,          form width in pixels            */
/*              b_width,        border width in pixels          */
/*              b_color,        border color                    */
/*              bk_color,       background color                */
/*    EXAMPLE :                                                 */
/*                                                              */
/*     ( XCInit "climate" "/u/grooper/kpohl/mycolors" 10 10     */
/*                 500 500 2 "red" "wheat                       */

PUBLIC int
XCAutoABSForm();
/*****************************************************************
* FUNCTION: Creates an automatic base form with upper left-hand  *
*             coordinates of ( x, y )                            *
*           This type of ABS Form AUTOMATICALLY captures the     *
*           specific window's image after each graphics call     *
*           to the server which alters the window's graphics.    *
*           Therefore, window image 'refreshing' is completely   *
*           transparent to the client. No call to XCSaveImage()  *
*           need ever be made. If you are unsure whether to use  *
*           an AUTOMATIC ABS Form or a MANUAL ABS Form using     *
*           an AUTOMATIC ABS Form would be safest.               *
* RETURNS:  WID of the FORM                                      *
*****************************************************************/
/*          x,              x pos. of upper left-hand corner    */
/*          y,              y pos. of upper left-hand corner    */
/*          height,         form height in pixels               */
/*          width,          form width in pixels                */
/*          b_width,        border width in pixels              */
/*          b_color,        border color                        */
/*          bk_color,       background color                    */
```

```
/*     EXAMPLE :                                                    */
/*                                                                  */
/*          ( XCAutoABSForm 50 50 400 400 2 "red" "wheat" )         */

PUBLIC int
XCManABSForm();
/*******************************************************************
* FUNCTION: Creates a Manual base form with upper left-hand       *
*               coordinates of ( x, y ).                          *
*           This type of ABSForm does not automatically save the  *
*           window image after primitive graphics calls to the    *
*           Server. Using XCManABSForm() allows a series of       *
*           graphic images to be drawn in quick succession        *
*           without the delay of having the Server save the       *
*           new image each time. However, in order for the imag   *
*           to be automatically 'refreshed' upon Exposure         *
*           a call to XCSaveImage() must be each time the         *
*           client wants the current image saved.                 *
* RETURNS:  WID of the FORM                                       *
*******************************************************************/
/*           x,              x pos. of upper left-hand corner     */
/*           y,              y pos. of upper left-hand corner     */
/*           height,         form height in pixels                */
/*           width,          form width in pixels                 */
/*           b_width,        border width in pixels               */
/*           b_color,        border color                         */
/*           bk_color,       background color                     */
/*     EXAMPLE :                                                  */
/*                                                                */
/*          ( XCManABSForm 50 50 400 400 2 "red" "wheat" )        */

PUBLIC int
XCAutoForm();
/*******************************************************************
* FUNCTION: Creates an automatic base form with the               *
*               corresponding adjacencies                         *
*           This type of Form AUTOMATICALLY captures the          *
*           specific window's image after each graphics call.     *
*                                                                 *
*           Therefore, window image 'refreshing' is completely    *
*           transparent to the client. No call to XCSaveImage()   *
*           need ever be made. If you are unsure whether to use   *
*           an AUTOMATIC Form or a MANUAL Form using              *
*           an AUTOMATIC Form would be safest.                    *
* RETURNS:  WID of the FORM                                       *
*******************************************************************/
/*           h_adj,          WID of horiz. adjacency              */
/*           v_adj,          WID of vert. adjacency               */
/*           b_width,        border width in pixels               */
/*           height,         form height in pixels                */
/*           width,          form width in pixels                 */
/*           b_color,        border color                         */
/*           bk_color,       background color                     */
```

```
/*    EXAMPLE :                                                    */
/*                                                                 */
/*        ( XCAutoForm 0 ?banner_WID 2 400 400 "red" "wheat" )     */

PUBLIC int
XCManForm();
/*****************************************************************
* FUNCTION: Creates a manual base form with the corresponding     *
*               adjacencies.                                      *
*                                                                 *
*           This type of Form does not automatically save the     *
*           window image after primitive graphics calls to the    *
*           Server. Using XCManForm() allows a series of          *
*           graphic images to be drawn in quick succession        *
*           without the delay of having the Server save the       *
*           new image each time. However, in order for the imag   *
*           to be automatically 'refreshed' upon Exposure,        *
*           a call to XCSaveImage() must be each time the         *
*           client wants the current image saved.                 *
* RETURNS:  WID of the FORM                                       *
*****************************************************************/
/*          h_adj,          WID of horiz. adjacency              */
/*          v_adj,          WID of vert. adjacency               */
/*          b_width,        border width in pixels               */
/*          height,         form height in pixels                */
/*          width,          form width in pixels                 */
/*          b_color,        border color                         */
/*          bk_color,       background color                     */
/*    EXAMPLE :                                                  */
/*                                                               */
/*        ( XCManForm 0 ?banner_WID 2 400 400 "red" "wheat" )    */

PUBLIC int
XCSaveImage();
/*****************************************************************
* FUNCTION: Saves the specific base form's image for exposure     *
*              'refreshing'. This call is used with MANUAL forms  *
* RETURNS:  NOTHING                                               *
*****************************************************************/
/*          wid             WID of its base form                 */
/*    EXAMPLE :                                                  */
/*                                                               */
/*        ( XCSaveImage ?ManForm1 )                              */

PUBLIC int
XCMenu();
/*****************************************************************
* FUNCTION: Creates a menu with the corresponding adjacencies     *
*               and characteristics (ie. color, border width)     *
*                                                                 *
* RETURNS:  WID of the MENU                                       *
*****************************************************************/
/*          base_form,      WID of its base form                 */
/*          h_adj,          WID of horiz. adjacency              */
/*          v_adj,          WID of vert. adjacency               */
```

```
/*          b_width,        border width in pixels              */
/*          b_color,        border color                        */
/*          t_color,        text color                          */
/*          bk_color,       background color                    */
/*          b_type,         (buttons) 0 = vert. // 1 = horiz.   */
/*          max_buttons,    # of buttons in V or H direction    */
/*          strgs,          multi-field fact of strings         */
/*    EXAMPLE :                                                 */
/*                                                              */
/*     (deffacts s1 (strgs "options" "search" "insert" "quit")) */
/*                                                              */
/*     ( XCMenu ?form2_WID 0 ?label_WID 2 "red" "white" "black" */
/*                    0 2 $?menu_strgs )                        */

PUBLIC int
XCPopup();
/*****************************************************************
* FUNCTION: Creates a popup menu with corresponding adjacencies  *
*               and characteristics (ie. color, border width)    *
*                                                                *
* RETURNS:  WID of the POPUP MENU                                *
*****************************************************************/
/*          base_form,      WID of its base form                */
/*          parent,         WID of its parent                   */
/*          popoff,         WID of the widget its popped off    */
/*          offset,         The button number to pop off of.    */
/*                              This is only applicable when    */
/*                              popping off menus. Sending in a */
/*                              ZERO value (ie. 0 ) indicates   */
/*                              that you will popoff something  */
/*                              other than a menu. In that case */
/*                              offset will be ignored.         */
/*          b_color,        border color                        */
/*          t_color,        text color                          */
/*          bk_color,       background color                    */
/*          b_width,        border width in pixels              */
/*          m_type,         (menu)    0 = vert. // 1 = horiz.   */
/*          b_type,         (buttons) 0 = vert. // 1 = horiz.   */
/*          max_buttons,    # of buttons in V or H direction    */
/*          strgs,          multi-field fact of strings         */
/*    EXAMPLE :                                                 */
/*                                                              */
/*     (deffacts s1 (strgs "options" "search" "insert" "quit")) */
/*                                                              */
/*     ( XCPopup ?form1 ?menu1 ?menu1 "blue" "white" "black"    */
/*                    2 1 0 2 $?menu_strgs )                    */

PUBLIC int
XCBanner();
/*****************************************************************
* FUNCTION: Creates a banner in the corresponding base form      *
* RETURNS:  WID of the BANNER                                    *
*****************************************************************/
/*          base_form,      WID of its base form                */
/*          b_strg,         banner string                       */
```

```
/*          b_width,        border width in pixels              */
/*          pix_size,       length in pixels                    */
/*          t_color,        text color                          */
/*          bk_color,       background color                    */
/*          font_style,     font style                          */
/*    EXAMPLE :                                                 */
/*                                                              */
/*     ( XCBanner ?form3 "-> MENU <-" 2 500 "magenta" "black"   */
/*                    "vr-40" )                                 */

PUBLIC int
XCLabel();
/*****************************************************************
* FUNCTION: Creates a label with the corresponding adjacencies   *
* RETURNS:  WID of the LABEL                                     *
*****************************************************************/
/*          base_form,      WID of its base form                */
/*          h_adj,          WID of horiz. adjacency             */
/*          v_adj,          WID of vert. adjacency              */
/*          l_strg,         banner string                       */
/*          b_width,        border width in pixels              */
/*          t_color,        text color                          */
/*          bk_color,       background color                    */
/*          font_style,     font style                          */
/*    EXAMPLE :                                                 */
/*                                                              */
/*     ( XCLabel ?form3 ?menu1 ?banner2 "-> MENU <-" 2 "black"  */
/*              "red" "Rom10.500" )                             */

PUBLIC HASH_PTR
XCReadMenu();
/*****************************************************************
* FUNCTION: Performs I/O through a menu. Allows the user to      *
*               select a button in the given menu                *
* RETURNS:  the string corresponding to the chosen button        *
*****************************************************************/
/*          menu,           WID of menu to be read              */
/*    EXAMPLE :                                                 */
/*                                                              */
/*        ( XCReadMenu ?popup3 )                                */

PUBLIC HASH_PTR
XCDialog();
/*****************************************************************
* FUNCTION: Creates a dialog box for input/output                *
* RETURNS:  the input string                                     *
*****************************************************************/
/*          base_form,      WID of its base form                */
/*          parent,         WID of its parent                   */
/*          popoff,         WID of the widget its popped off    */
/*          msg,            output message                      */
/*          cols,           number of text columns              */
/*          b_width,        border width in pixels              */
/*          d_type,         (dialog)  0 = vert. // 1 = horiz.   */
```

```
/*    EXAMPLE :                                                    */
/*                                                                 */
/* (XCDialog ?form2 ?form2 ?menu1 "Enter your name..." 30 2 1 ) */

PUBLIC int
XCDelete();
/*******************************************************************
* FUNCTION: Deletes the given widget                               *
* RETURNS:  NOTHING                                                *
*******************************************************************/
/*          wid,            WID of widget to delete                */
/*    EXAMPLE :                                                    */
/*                                                                 */
/*        ( XCDelete ?popup3 )                                     */

PUBLIC int
XCFontStyle();
/*******************************************************************
* FUNCTION:  Sets the font style in the given FORM                 *
* RETURNS:  NOTHING                                                *
*******************************************************************/
/*          wid,            WID of the FORM                        */
/*          font,           font style                             */
/*    EXAMPLE :                                                    */
/*                                                                 */
/*        ( XCFontStyle ?ABSform3 "Itl14.500" )                    */

PUBLIC int
XCSetColor();
/*******************************************************************
* FUNCTION: Sets the drawing color in the given FORM               *
* RETURN:   NOTHING                                                *
*******************************************************************/
/*          wid,            WID of the FORM                        */
/*          d_color,        drawing color                          */
/*    EXAMPLE :                                                    */
/*                                                                 */
/*        ( XCSetColor ?form2 "magenta" )                          */

PUBLIC int
XCChangeBkgColor();
/*******************************************************************
* FUNCTION: Changes the background color in the given Widget       *
*               NOTE : Color changes immediately upon call         *
*                                                                  *
* RETURNS:  NOTHING                                                *
*******************************************************************/
/*          wid,            WID of the FORM                        */
/*          color,          background color                       */
/*    EXAMPLE :                                                    */
/*                                                                 */
/*        ( XCChangeBkgColor ?label1 "magenta" )                   */
```

```
PUBLIC int
XCLineStyle();
/*****************************************************************
* FUNCTION: Sets the drawing line style in the given FORM        *
*                                                                *
* RETURNS:  NOTHING                                              *
*****************************************************************/
/*          wid,            WID of the FORM                      */
/*          l_width,        line width in pixels                 */
/*          l_style,        line style                           */
/*                               0 = LineSolid                   */
/*                               1 = LineOnOffDash               */
/*                               2 = LineDoubleDash              */
/*          c_style,        cap style ->BEWARE OF CapRound<-     */
/*                               0 = CapNotLast                  */
/*                               1 = CapButt                     */
/*                               2 = CapRound                    */
/*                               3 = CapProjecting               */
/*          j_style,        join style                           */
/*                               0 = JoinMiter                   */
/*                               1 = JoinRound                   */
/*                               2 = JoinBevel                   */
/*    EXAMPLE :                                                  */
/*                                                               */
/*        ( XCLineStyle ?form1 5 2 0 1 )                         */

PUBLIC int
XCLines();
/*****************************************************************
* FUNCTION: Draws multiple connected lines in the given FORM     *
*                                                                *
* RETURNS:  NOTHING                                              *
*****************************************************************/
/*          wid,            WID of the FORM                      */
/*          points,         multi-field fact of (x,y) pairs      */
/*    EXAMPLE :                                                  */
/*                                                               */
/*        FACT FORMAT : x y                                      */
/*                                                               */
/*        (deffacts p1 (pts 10 20   300 200   400 10   50 50))  */
/*                                                               */
/*        ( XCLines ?form2 $?points )                            */

PUBLIC int
XCSegments();
/*****************************************************************
* FUNCTION: Draws multiple line segments in the FORM             *
*             NOTE: XWindows has a MAJOR problem with drawing    *
*                   diagonal line segments when the CAP STYLE is *
*                   set to CapRound...( CapRound is suspect for  *
*                   causing other problems as well..BE WARNED )  *
* RETURNS:  NOTHING                                              *
*****************************************************************/
/*          wid,            WID of the FORM                      */
/*          segments,       multi-field fact of (x,y) pairs      */
```

```
/*    EXAMPLE :                                                    */
/*                                                                 */
/*        FACT FORMAT : x1 y1 x2 y2                                */
/*                                                                 */
/*        (deffacts s1 (segs 10 20 300 200   400 10 50 50))        */
/*                                                                 */
/*        ( XCSegments ?form3 $?segments )                         */

PUBLIC int
XCArcs();
/*****************************************************************
* FUNCTION: Draws multiple arcs in the given FORM                 *
* RETURNS:  NOTHING                                               *
*****************************************************************/
/*          wid,            WID of the FORM                      */
/*          arcs,           multi-field fact of arc data         */
/*    EXAMPLE :                                                  */
/*                                                               */
/*        FACT FORMAT : x y width height start_angle             */
/*                      continuation_angle                       */
/*                                                               */
/* (deffacts a1 (arcs 10 20 100 200 0 360 10 100 20 20 45 180)) */
/*                                                               */
/*        ( XCArcs ?form1 $?arcs )                               */

PUBLIC int
XCFillArcs();
/*****************************************************************
* FUNCTION: Draws multiple filled arcs in the given FORM          *
*           NOTE: In order to allow for dynamic string passing    *
*                 the size of each string (color) is determined   *
*                 at run time rather than compile time. With      *
*                 this in mind the colors argument should be      *
*                 defined as an array of pointers to char's and   *
*                 not as a static two dimensional array of chars  *
*                 (ie. malloc space for each color string)        *
* RETURNS:  NOTHING                                               *
*****************************************************************/
/*          wid,            WID of the FORM                      */
/*          arcs,           multi-field fact of arc data         */
/*          colors,         multi-field fact of colors           */
/*    EXAMPLE :                                                  */
/*                                                               */
/*       FACT FORMAT : x y width height start_angle              */
/*                     continuation_angle                        */
/*                                                               */
/* (deffacts a1 (arcs 10 20 100 200 0 360 10 100 20 20 45 180)) */
/*        (deffacts c1 (cols "red" "green"))                     */
/*                                                               */
/*        ( XCFillArcs ?form1 $?arcs $?colors )                  */
```

```
PUBLIC int
XCFillPolys();
/*****************************************************************
* FUNCTION: Fills the region(s) enclosed by the given polygons   *
* RETURNS:  NOTHING                                              *
*****************************************************************/
/*          wid,            WID of the FORM                      */
/*          polypoints,     multi-field fact of (x,y) pairs      */
/*          fill_color,     filling color for the polygon        */
/*    NOTE:                                                      */
/*    You can specify as many polygons as you wish in one call. */
/*                                                               */
/*    EXAMPLE :                                                  */
/*                                                               */
/*        FACT FORMAT : x1 y1   x2 y2 ... xn yn                  */
/*                                                               */
/* (deffacts fp (fpts 10 20  100 200  10 360  10 10   10 20))    */
/*                                                               */
/* (XCFilledPolys ?form2 $?polypoints1 "red" polypoints2 "blue")*/
PUBLIC int
XCDrawStrings();
/*****************************************************************
* FUNCTION: Draws multiple strings at their corresponding        *
*               ( x,y ) positions                                *
* RETURNS:  NOTHING                                              *
*****************************************************************/
/*          wid,            WID of the FORM                      */
/*          strgs,          multi-field fact of strings          */
/*          points          multi-field fact of (x,y) pairs      */
/*    EXAMPLE :                                                  */
/*                                                               */
/*        FACT FORMAT : x y                                      */
/*                                                               */
/*   (deffacts p1 (pts 10 20   300 200   400 10   50 50))       */
/*   (deffacts s1 (strgs "Hello world" "34.5" "ohoh" "byebye")   */
/*                                                               */
/*        ( XCDrawStrings ?form1 $?strgs1 $?strg_points )       */

PUBLIC int
XCQuit();
/*****************************************************************
* FUNCTION: Disconnects the client from the server               *
*           ( Ends the session )                                 *
* RETURNS:  NOTHING                                              *
*****************************************************************/
/*    EXAMPLE :                                                  */
/*                                                               */
/*        ( XCQuit )                                             */

PUBLIC int
XCStopServer();
/*****************************************************************
* FUNCTION: Terminates the mother Server...Each client must      *
*              still make a call to XCQuit() to end its session  *
*              nicely                                           *
* RETURNS:  NOTHING                                              *
*****************************************************************/
/*    EXAMPLE :                                                  */
/*                                                               */
/*        ( XCStopServer )                                       */
```

The GXI message structure, which is passed between the clients and the server, follows. It contains all of the information used to define a GXI client request.

```
/*****************************************************************
   CMD and RESULT structure
*****************************************************************/
typedef struct {
   int   cmd_code;
   int   strg_flag;            /* TRUE if sending STRINGS       */
   int   pts_flag;             /* TRUE if sending POINTS        */
   int   segs_flag;            /* TRUE if sending SEGMENTS      */
   int   arcs_flag;            /* TRUE if sending ARCS          */
   int   fpoly_flag;           /* TRUE if sending FILLEDPOLYS   */

   int   strg_msg_size;        /* size of the strg msg          */
   int   num_strgs;            /* number of passed strings      */
   char   **strgs;             /* ptr to array of strings       */

   int   pts_buf_size;         /* size of the points buffer     */
   int   npoints;              /* number of passed points       */
   XC_Point *points;           /* pointer to XC_Point strc      */

   int   segs_buf_size;        /* size of the segments buffer   */
   int   nsegments;            /* number of passed segments     */
   XC_Segment *segments;       /* pointer to XC_Segment strc    */

   int   arcs_buf_size;        /* size of the arcs buffer       */
   int   narcs;                /* number of passed arcs         */
   XC_Arc *arcs;               /* pointer to XC_Arc strc        */

   int   fpoly_buf_size;       /* size of the filled polys      */
   int   nfilledpolys;         /* number of passed filled polys */
   XC_FilledPoly *filledpolys; /* ptr to XC_FilledPoly          */

   union {
      struct {
         char XCdisplay[MAX_DNAME_SIZE;   /* client's displ.    */
         char color_file[MAX_FNAME_SIZE]; /* name of color file */
         int   height;                    /* height             */
         int   width;                     /* width              */
         int   b_width;                   /* border width       */
         char  b_color [ MAX_SIZE ];      /* border color       */
         char   bk_color[ MAX_SIZE ];     /* background color   */
      } _INIT;
      struct {
         int   height;                    /* height             */
         int   width;                     /* width              */
```

```
   int   b_width;                /* border width          */
   char  b_color [ MAX_SIZE ];   /* border color          */
   char   bk_color[ MAX_SIZE ];  /* background color      */
} _ABSFORM;
struct {
   int    h_adj;                 /* WID                   */
   int    v_adj;                 /* WID                   */
   int   b_width;                /* border width          */
   int   height;                 /* height                */
   int   width;                  /* width                 */
   char  b_color [ MAX_SIZE ];   /* border color          */
   char   bk_color[ MAX_SIZE ];  /* background color      */
} _FORM;
struct {
   int    wid;                   /* WID                   */
} _SAVEIMAGE;
struct {
   int    base_form;             /* WID                   */
   int   h_adj;                  /* WID                   */
   int   v_adj;                  /* WID                   */
   int   b_width;                /* border width          */
   char  t_color[ MAX_SIZE ];    /* text color            */
   char  b_color[ MAX_SIZE ];    /* border color          */
   char  bk_color[ MAX_SIZE ];   /* background color      */
   int   b_type;                 /* 0 = V // 1 = H BUTTON */
   int   max_buttons;            /* # of buttons V or H   */
} _MENU;
struct {                         /* WID                   */
   int    parent;                /* WID                   */
   int   popoff;                 /* WID   (MENU)          */
   char  t_color[ MAX_SIZE ];    /* text color            */
   char  b_color[ MAX_SIZE ];    /* border color          */
   char  bk_color[ MAX_SIZE ];   /* background color      */
   int   b_width;                /* border width          */
   int   m_type;                 /* 0 = V // 1 = H MENU   */
   int   b_type;                 /* 0 = V // 1 = H BUTTON */
   int   max_buttons;            /* # of buttons V or H   */
   int   offset;                 /* button to pop off of  */
} _POPUP;
struct {
   int    base_form;             /* WID                   */
   int   b_width;                /* border width          */
   int   pix_size;               /* banner length         */
   char  t_color[ MAX_SIZE ];    /* text color            */
   char  bk_color[ MAX_SIZE ];   /* background color      */
   char  font_style[ MAX_SIZE ]; /* banner font style     */
} _BANNER;
struct {
   int    base_form;             /* WID                   */
   int   h_adj;                  /* WID                   */
   int   v_adj;                  /* WID                   */
   int   b_width;                /* border width          */
   char  t_color[ MAX_SIZE ];    /* text color            */
   char  bk_color[ MAX_SIZE ];   /* background color      */
   char  font_style[ MAX_SIZE ]; /* label font style      */
```

```
        } _LABEL;
        struct {
           int    wid;                    /* WID                    */
        } _DELETE;
        struct {
           int    menu;                   /* WID                    */
        } _READMENU;
        struct {
           int    base_form;              /* WID                    */
           int    parent;                 /* WID                    */
           int    popoff;                 /* WID                    */
           int   cols;                    /* text columns           */
           int   input;                   /* 1 = input/output       */
           int   b_width;                 /* border width           */
           int   d_type;                  /* 0=VERT. // 1=HORIZ.    */
        } _DIALOG;
        struct {
           int    wid;                    /* WID                    */
        } _DRAWSTRINGS;
        struct {
           int    wid;                    /* WID                    */
           char  font [ MAX_SIZE ];       /* font style             */
        } _FONTSTYLE;
        struct {
           int    wid;                    /* WID                    */
           char  d_color [ MAX_SIZE ];    /* drawing color          */
        } _SETCOLOR;
        struct {
           int    wid;                    /* WID                    */
           char  color [ MAX_SIZE ];      /* background color       */
        } _CHGBKGCOLOR;
        struct {
           int    wid;                    /* WID                    */
           int   l_width;                 /* line width             */
           int   l_style;                 /* line style             */
           int   c_style;                 /* cap style              */
           int   j_style;                 /* join style             */
        } _LINESTYLE;
        struct {
           int   wid;                     /* WID                    */
        } _LINES;
        struct {
           int   wid;                     /* WID                    */
        } _SEGS;
        struct {
           int   wid;                     /* WID                    */
        } _ARCS;
        struct {
           int   wid;                     /* WID                    */
        } _FILLARCS;
        struct {
           int    wid;                    /* WID                    */
        } _FILLPOLYS;
        struct {
           int    wid;                    /* WID                    */

        } _NEW_WID;
     } data;
  } cmd_type;
```

9. Appendix C: MMS Functional Interface

Part (1): FUNCTIONAL INTERFACE

```
        BDT_Int MMSConnect(
                MMSClient ClientName
            );
ASSUMPTIONS:    --
    EFFECTS:    Registers ClientName with the MMS and
                issues ClientName basic authorization
 RET. VALUE:    If successful, returns MMS_SUCCESS
                else, returns an appropriate MMS Error Code
ERROR CODES:    MMS_ALREADY_REGISTERED,
                MMS_SYSTEM_ERROR, ...
 REFERENCES:    MMSDisconnect()

        BDT_Int MMSDisconnect(
                MMSClient ClientName
            );
ASSUMPTIONS:    --
    EFFECTS:    Unregisters ClientName with the MMS
 RET. VALUE:    If successful, returns MMS_SUCCESS
                else, returns an appropriate MMS Error Code
ERROR CODES:    MMS_UNATHORIZED_ACTION,
                MMS_UNKNOWN_CLIENT,
                MMS_SYSTEM_ERROR, ...
 REFERENCES:    MMSConnect()

        BDT_Int MMSDisableMMSAuthorization(
                MMSClient ClientName
            );
ASSUMPTIONS:    The issuing client is registered with
                MMS. ClientName intends to take respons-
                ibility for authorization generation
    EFFECTS:    Disables MMS authorization generation
                facilities. Note that authorization checking
                on MMS functions is still performed. The
                default envirnonment enables MMS author-
                ization generation
 RET. VALUE:    If successful, returns MMS_SUCCESS
                else, returns an appropriate MMS Error Code
ERROR CODES:    MMS_ALREADY_DISABLED,
                MMS_UNKNOWN_CLIENT,
                MMS_SYSTEM_ERROR
 REFERENCES:    MMSEnableMMSAuthorization()

        BDT_Int MMSEnableMMSAuthorization(
                MMSClient ClientName
            );
ASSUMPTIONS:    The issuing client is registered with
                MMS. ClientName initially disabled MMS
                authorization generation
    EFFECTS:    Enables MMS authorization generation
```

```
                facilities. The default environment enables
                MMS authorization generation
 RET. VALUE:    If successful, returns MMS_SUCCESS
                else, returns an appropriate MMS Error Code
ERROR CODES:    MMS_UNAUTHORIZED_ACTION,
                MMS_ALREADY_ENABLED,
                MMS_UNKNOWN_CLIENT,
                MMS_SYSTEM_ERROR
 REFERENCES:    MMSDisableMMSAuthorization()

        BDT_Int MMSRegisterMsgHandler(
                BDT_Int MsgType,
                MMSMsgHandler ClientMsgHandler
              );
ASSUMPTIONS:    The issuing client is registered with
                MMS. MsgType is a valid MMS message type
    EFFECTS:    Associates ClientMsgHandler with messages
                of type MsgType. ClientMsgHandler will be
                called each time a message of type MsgType
                is received
 RET. VALUE:    If successful, returns MMS_SUCCESS
                else, returns an appropriate MMS Error Code
ERROR CODES:    MMS_ALREADY_REGISTERED,
                MMS_SYSTEM_ERROR
 REFERENCES:    MMSUnregisterMsgHandler(),
                MMSChangeMsgHandler(), MMSReceiveMsg(),
                MMSReceiveMsgType(),
                MMSReceiveMsgFromClient()

        BDT_Int MMSUnregisterMsgHandler(
                BDT_Int MsgType
              );
ASSUMPTIONS:    The issuing client is registered with
                MMS and has previously registered a message
                handler for messages of type MsgType.
                MsgType is a valid MMS message type
    EFFECTS:    Disassociates the client message handler
                with messages of type MsgType.
 RET. VALUE:    If successful, returns MMS_SUCCESS
                else, returns an appropriate MMS Error Code
ERROR CODES:    MMS_SYSTEM_ERROR
 REFERENCES:    MMSUnregisterMsgHandler(),
                MMSChangeMsgHandler(), MMSReceiveMsg(),
                MMSReceiveMsgType(),
                MMSReceiveMsgFromClient()

     MsgHandler MMSChangeMsgHandler(
                BDT_Int MsgType,
                MMSMsgHandler ClientMsgHandler
              );
ASSUMPTIONS:    The issuing client is registered with
                MMS and has previously registered a message
                handler for messages of type MsgType.
                MsgType is a valid MMS message type
```

```
    EFFECTS:     Associates the ClientMsgHandler
                 with messages of type MsgType
 RET. VALUE:     If successful, returns the old client message handler
                 else, returns an appropriate MMS Error Code
ERROR CODES:     MMS_SYSTEM_ERROR
 REFERENCES:     MMSUnregisterMsgHandler(),
                 MMSChangeMsgHandler(), MMSReceiveMsg(),
                 MMSReceiveMsgType(),
                 MMSReceiveMsgFromClient()

        BDT_Int MMSSendMsg(
                 MMSMessage  ClientMsg;
              );
ASSUMPTIONS:     The issuing client is registered with MMS
    EFFECTS:     ClientMessage is sent to the TargetClientY
                 Note that this function call returns as soon
                 as the message has left the senders process
                 environment. Therefore, there may be no
                 attempt to notify the sending client of
                 any failure to reach the TargetClient
 RET. VALUE:     If successful, returns MMS_SUCCESS
                 else, returns an appropriate MMS Error Code
ERROR CODES:     MMS_UNKNOWN_CLIENT,
                 MMS_SYSTEM_ERROR
 REFERENCES:     MMSSendAndAckMsg(), MMSBroadcastMsg(),
                 MMSReceiveMsg(),
                 MMSReceiveMsgType(),
                 MMSReceiveMsgFromClient()

        BDT_Int MMSSendAndAckMsg(
                 MMSMessage  ClientMsg;
              );
ASSUMPTIONS:     The issuing client is registered with MMS
    EFFECTS:     ClientMsg is sent to the TargetClient.
                 Note that in this function, the MMS follows
                 the status of the ClientMsg until it enters
                 the MMS section of the TargetClient's process
                 environment or until an error occurs. In
                 either case, the MMS returns the final status
                 of the transaction as a return value. Note
                 also that the MMS imposes a MMS defined time
                 limit on determining the final status of the
                 ClientMsg
 RET. VALUE:     If successful, returns MMS_SUCCESS
                 else, returns an appropriate MMS Error Code
ERROR CODES:     MMS_UNKNOWN_CLIENT,
                 MMS_TIMEOUT,
                 MMS_SYSTEM_ERROR
 REFERENCES:     MMSSendMsg(), MMSBroadcastMsg(),
                 MMSReceiveMsg(),
                 MMSReceiveMsgType(),
                 MMSReceiveMsgFromClient()

        BDT_Int MMSBroadcastMsg(
```

```
                MMSMessage  ClientMsg;
            );
ASSUMPTIONS:    The issuing client is registered with MMS
    EFFECTS:    ClientMessage is sent to all clients
                currently registered with MMS.
                Note that this function call returns as soon
                as all messages have left the senders process
                environment. Therefore, there may be no
                attempt to notify the sending client of
                any failure to reach the target clients.
                Note that the TargetClient entry in
                ClientMsg is ignored
 RET. VALUE:    If successful, returns MMS_SUCCESS
                else, returns an appropriate MMS Error Code
ERROR CODES:    MMS_UNAUTHORIZED_ACTION,
                MMS_SYSTEM_ERROR
 REFERENCES:    MMSSendMsg(), MMSSendAndAckMsg(),
                MMSReceiveMsg(),
                MMSReceiveMsgType(),
                MMSReceiveMsgFromClient()

        BDT_Int MMSReceiveMsg(
            );
ASSUMPTIONS:    The issuing client is registered with MMS
    EFFECTS:    The MMS Message Queue is checked
                for a pending message. If one does not
                exist, the client is put to sleep until
                it receives a message from any other MMS
                client. In either case, upon reception of
                a message, the clients associated message
                handler is called. When the handler
                completes, MMSReceiveMsg() returns
 RET. VALUE:    If successful, returns MMS_SUCCESS
                else, returns an appropriate MMS Error Code
ERROR CODES:    MMS_SYSTEM_ERROR
 REFERENCES:    MMSSendMsg(), MMSSendAndAckMsg(),
                MMSBroadcastMsg(),
                MMSReceiveMsgType(),
                MMSReceiveMsgFromClient()

        BDT_Int MMSReceiveMsgType(
                BDT_Int MsgType;
            );
ASSUMPTIONS:    The issuing client is registered with MMS.
                MsgType is a valid MMS message type
    EFFECTS:    The MMS Message Queue is checked
                for a pending message of type MsgType.
                If one does not exist, the client is
                put to sleep until it receives a message
                of type MsgType from any other MMS client.
                In either case, upon reception of a
                message, the clients associated message
                handler is called. When the handler
                completes, MMSReceiveMsgType() returns
```

```
 RET. VALUE:     If successful, returns MMS_SUCCESS
                 else, returns an appropriate MMS Error Code
ERROR CODES:     MMS_SYSTEM_ERROR
 REFERENCES:     MMSSendMsg(), MMSSendAndAckMsg(),
                 MMSBroadcastMsg(),
                 MMSReceiveMsg(),
                 MMSReceiveMsgFromClient()

        BDT_Int MMSReceiveMsgFromClient(
                 MMSClient  ClientName;
              );
ASSUMPTIONS:     The issuing client is registered with MMS
    EFFECTS:     The MMS Message Queue is checked
                 for a pending message from ClientName.
                 If one does not exist, the client
                 is put to sleep until it receives a message
                 from ClientName. In either case, upon
                 reception of a message, the clients
                 associated message handler is called.
                 When the handler completes,
                 MMSReceiveMsgFromClient() returns.
                 Note that this function call creates
                 the potential for a DEADLOCK. The MMS
                 takes no responsibility for such an
                 occurrance. Clients may use this
                 function call at their own discretion
 RET. VALUE:     If successful, returns MMS_SUCCESS
                 else, returns an appropriate MMS Error Code
ERROR CODES:     MMS_UNAUTHORIZED_ACTION,
                 MMS_UNKNOWN_CLIENT,
                 MMS_SYSTEM_ERROR
 REFERENCES:     MMSSendMsg(), MMSSendAndAckMsg(),
                 MMSBroadcastMsg(),
                 MMSReceiveMsg(), MMSReceiveMsgType()

        BDT_Int MMSDisableReceiveMsg(
              );
ASSUMPTIONS:     The issuing client is registered with MMS
    EFFECTS:     Temporarily disables the transparent
                 receiving of messages by the the issuing
                 client. All incoming messages are placed
                 in the MMS Message Queue in chronological
                 order relative to the time at which they
                 entered the receiving process environment.
                 To preserve the "real time" characterists
                 of the MMS, disabling of receive message
                 should be performed at an absolute
                 minimum.
                 Note, the default environment supports
                 receive message enabled.
 RET. VALUE:     If successful, returns MMS_SUCCESS
                 else, returns an appropriate MMS Error Code
ERROR CODES:     MMS_ALREADY_DISABLED,
                 MMS_SYSTEM_ERROR
```

```
 REFERENCES:     MMSEnableReceiveMsg()

         BDT_Int MMSEnableReceiveMsg(
             );
ASSUMPTIONS:     The issuing client is registered with MMS
    EFFECTS:     Enables the transparent receiving of
                 messages by the the issuing client.
                 Any pending messages stored in the MMS
                 Message Queue are dispensed in
                 chronological order relative to the
                 time they entered the receiving time
                 process environment using the MMS
                 message handler mechanism.
                 Note, the default environment supports
                 receive message enabled.
 RET. VALUE:     If successful, returns MMS_SUCCESS
                 else, returns an appropriate MMS Error Code
ERROR CODES:     MMS_ALREADY_ENABLED,
                 MMS_SYSTEM_ERROR
 REFERENCES:     MMSDisableReceiveMsg()

         BDT_Int MMSGetMMSClients(
                 BDT_Int *NumMMSClientsPtr;
                 MMSClientPtr  *MMSClients
             );
ASSUMPTIONS:     The issuing client is registered with MMS
    EFFECTS:     NumMMSClientsPtr is set to point to the
                 number of current MMS clients. MMSClients
                 is set to point to an array of current MMS
                 clients. Note, it is the clients
                 responsibility to free these two pointers
                 when they are no longer needed
 RET. VALUE:     If successful, returns MMS_SUCCESS
                 else, returns an appropriate MMS Error Code
ERROR CODES:     MMS_UNAUTHORIZED_ACTION,
                 MMS_SYSTEM_ERROR
 REFERENCES:     --

      MMSBoolean MMSPendingMsgsForClient(
                 MMSClient  ClientName;
             );
ASSUMPTIONS:     The issuing client is registered with MMS
    EFFECTS:     Checks the entire MMS for the occurrance
                 of a message in transit to, or in the
                 MMS Message Queue of, ClientName
 RET. VALUE:     If found, returns MMS_TRUE,
                 if not found, returns MMS_FALSE,
                 else, returns an appropriate MMS Error Code
ERROR CODES:     MMS_UNAUTHORIZED_ACTION,
                 MMS_UNKNOWN_CLIENT,
                 MMS_SYSTEM_ERROR
 REFERENCES:     --
```

Part (2): FACT MANAGER FUNCTIONAL INTERFACE

```
******  CONSTRUCTORS and DESTRUCTORS  ******
FM_FactRef        FMCreateFact()
FM_FactRef        FMCopyFact( FM_FactRef FactRef )
BDT_Void          FMDestroyFact( FM_FactRef FactRef )
BDT_Void          FMDestroyField( FM_FieldRef FieldRef )

******  FACT BUILDING  ******
FM_FactRef        FMAddFloat(
                    FM_FactRef FactRef,
                    BDT_Float Float
                  )
FM_FactRef        FMAddWord(
                    FM_FactRef FactRef,
                    BDT_Char *Word
                  )
FM_FactRef        FMAddStrg(
                    FM_FactRef FactRef,
                    BDT_Char *Strg
                  )
FM_FactRef        FMAddUserDefObj(
                    FM_FactRef FactRef,
                    BDT_Ref ObjRef,
                    BDT_Int ObjSize
                  )

******  FACT PARSING  ******
BDT_Void          FMResetCursor( FM_FactRef FactRef )
FM_FieldRef       FMGetNextField( FM_FactRef FactRef )
BDT_Int           FMGetFactSize( FM_FactRef FactRef )
BDT_Int           FMGetNumFactFields( FM_FactRef FactRef )

******  FACT DEBUGGING  ******
BDT_Int           FMPrintFact( FM_FactRef FactRef )

******  FACT SENDING  ******
BDT_Int           MASFSendFact(
                    MMSC_ClientRef DstClientRef,
                    FM_FactRef FactRef
                  )
BDT_Int           MASFMultiSendFact(
                    MMSC_ListRef DstClientListRef,
                    FM_FactRef FactRef
                  )
BDT_Int           MASFBroadcastFact(
                    FM_FactRef FactRef
                  )

******  FACT BUFFER SENDING  ******
BDT_Int           MASFSendBuffer(
                    MMSC_ClientRef DstClientRef,
```

```
                    FM_BufferRef BufferRef
                  )
BDT_Int           MASFMultiSendBuffer(
                    MMSC_ListRef DstClientListRef,
                    FM_BufferRef BufferRef
                  )
BDT_Int           MASFBroadcastBuffer(
                    FM_BufferRef BufferRef
                  )
```

10. Appendix D: CMS Functional Interface

Part (1): Basic Functional Interface

SESSION MANAGER:

```
**********************************************************************
**                                                                  **
**    FUNCTION       : CMS_IdentityRef CMSConnect()                 **
**    DESCRIPTION    :                                              **
**       Connects the caller to a multi-agent session.              **
**    Note, the default "transparent" receive mode is "transparent" **
**    receive DISABLED...                                           **
**                                                                  **
**    ARGUMENTS      :                                              **
**       char              *Name                                    **
**    RETURN VALUE   :                                              **
**       A reference to the object representation of the caller     **
**    or EM_ERROR_REF. Note, this object can be passed to other     **
**    clients as a way to address the caller directly...            **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**    FUNCTION       : int CMSDisconnect()                          **
**    DESCRIPTION    :                                              **
**       Disconnects the caller from a multi-agent session.         **
**                                                                  **
**    ARGUMENTS      :                                              **
**    RETURN VALUE   :                                              **
**       EM_SUCCESS or EM_ERROR                                     **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**    FUNCTION       : int CMSDestroyIdentity()                     **
**    DESCRIPTION    :                                              **
**       Destroys <IdRef>.                                          **
**                                                                  **
**    ARGUMENTS      :                                              **
**       CMS_IdentityRef   IdRef                                    **
**    RETURN VALUE   :                                              **
**       EM_SUCCESS or EM_ERROR                                     **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**    FUNCTION       : S_Boolean CMSIsSameIdentity()                **
**    DESCRIPTION    :                                              **
**       Determines if <Id1Ref> and <Id2Ref> are equal.             **
```

```
**                                                                    **
**   ARGUMENTS      :                                                 **
**      CMS_IdentityRef   Id1Ref                                      **
**      CMS_IdentityRef   Id2Ref                                      **
**   RETURN VALUE   :                                                 **
**      S_TRUE or S_FALSE                                             **
**                                                                    **
************************************************************************

************************************************************************
**                                                                    **
**   FUNCTION       : int CMSPutCreateIdentity()                      **
**   DESCRIPTION    :                                                 **
**      Places <IdRef> into <BufRef> with an object type              **
**   of CMS_CREATE_IDENTITY.                                          **
**                                                                    **
**   ARGUMENTS      :                                                 **
**      CMS_BufRef        BufRefRef                                   **
**      CMS_IdentityRef   IdRef                                       **
**   RETURN VALUE   :                                                 **
**      EM_SUCCESS or EM_ERROR                                        **
**                                                                    **
************************************************************************

************************************************************************
**                                                                    **
**   FUNCTION       : CMS_IdentityRef CMSGetIdentity()                **
**   DESCRIPTION    :                                                 **
**      Removes and returns the next object in <BufRef> as a          **
**   CMS_Identity.                                                    **
**                                                                    **
**   ARGUMENTS      :                                                 **
**      CMS_BufRef        BufRefRef                                   **
**   RETURN VALUE   :                                                 **
**      A reference to an identity or EM_ERROR_REF.                   **
**                                                                    **
************************************************************************

************************************************************************
**                                                                    **
**   FUNCTION       : void CMSPrintIdentity()                         **
**   DESCRIPTION    :                                                 **
**      Prints <IdRef> to stdout.                                     **
**                                                                    **
**   ARGUMENTS      :                                                 **
**      CMS_IdentityRef   IdRef                                       **
**   RETURN VALUE   :                                                 **
**                                                                    **
************************************************************************

************************************************************************
**                                                                    **
**   FUNCTION       : int CMSEnterGrp()                               **
**   DESCRIPTION    :                                                 **
```

```
**      Enrolls the caller into the group <GrpName>. Note that      **
**   the group name defined by SM_CMS_SYS_GRP_NAME ir reserved by   **
**   CMS.                                                           **
**                                                                  **
**   ARGUMENTS      :                                              **
**      char              *GrpName                                  **
**   RETURN VALUE   :                                              **
**      The caller's instance number within <GrpName> or EM_ERROR.  **
**   Instance numbers start at 0 and increment upward.              **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**   FUNCTION       : int CMSExitGrp()                             **
**   DESCRIPTION    :                                              **
**      Unenrolls the caller from the group <GrpName>.              **
**                                                                  **
**   ARGUMENTS      :                                              **
**      char              *GrpName                                  **
**   RETURN VALUE   :                                              **
**      EM_SUCCESS or EM_ERROR                                      **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**   FUNCTION       : int CMSSpawn()                               **
**   DESCRIPTION    :                                              **
**      Spawns <NumCopies> copies of <Task> in accordance with      **
**   the values of <How> and <where>.                               **
**                                                                  **
**   ARGUMENTS      :                                              **
**      char              *Task                                     **
**      int               NumCopies                                 **
**      CMS_How            How :                                    **
**                    CMS_TASK_DEFAULT- Target host is              **
**                                      determined by the           **
**                                      Session Manager. In         **
**                                      this case <Where> should    **
**                                      be NULL.                    **
**                       CMS_HOST    - Target host is               **
**                                      determined by <Where>.      **
**                       CMS_ARCH    - Target host is               **
**                                      determined by the           **
**                                      Session Manager. The        **
**                                      target host architecture    **
**                                      will be of type <Where>.    **
**                       CMS_DEBUG   - Target host is               **
**                                      determined by the           **
**                                      Session Manager. In         **
**                                      this case <Where> should    **
**                                      be NULL. All copies of      **
**                                      <Task> will be run in       **
```

```
**                                          the PVM debugger...          **
**                              CMS_TRACE   - Target host is             **
**                                          determined by the            **
**                                          Session Manager. In          **
**                                          this case <Where> should     **
**                                          be NULL. All copies of       **
**                                          <Task> will generate PVM     **
**                                          trace data...                **
**      char                *Where          - Examples :                 **
**                                          "moby.cadrc.calpoly.edu"     **
**                                          or "HPPA"                    **
**      char                **ArgV          - NULL terminated            **
**                                                                       **
**    RETURN VALUE   :                                                   **
**      The actual number of successfully spawned tasks or EM_ERROR.     **
**                                                                       **
***************************************************************************

***************************************************************************
**                                                                       **
**    FUNCTION       : int CMSSetDefaultMsgHandler()                     **
**    DESCRIPTION    :                                                   **
**      Associates the calling of <Handler> with the reception           **
**    of ANY message without an associated message handler.              **
**    <DataRef> is passed as an argument each time...                    **
**                                                                       **
**    <Handler> should have the following signature:                     **
**                                                                       **
**      void MyHandler( CMS_BufRef BufRef, S_VoidRef DataRef );          **
**                                                                       **
**    ARGUMENTS      :                                                   **
**      CMS_MsgHandler Handler                                           **
**      S_VoidRef      DataRef -- NULL if no data                        **
**    RETURN VALUE   :                                                   **
**      EM_SUCCESS or EM_ERROR                                           **
**                                                                       **
***************************************************************************

***************************************************************************
**                                                                       **
**    FUNCTION       : int CMSAddMsgHandler()                            **
**    DESCRIPTION    :                                                   **
**      Associates the calling of <Handler> with the reception           **
**    of messages of type <MsgType> passing <DataRef> as an              **
**    argument each time...                                              **
**                                                                       **
**    <Handler> should have the following signature:                     **
**                                                                       **
**      void MyHandler( CMS_BufRef BufRef, S_VoidRef DataRef );          **
**                                                                       **
**    ARGUMENTS      :                                                   **
**      int            MsgType -- MUST BE POSITIVE                       **
**      CMS_MsgHandler Handler                                           **
**      S_VoidRef      DataRef -- NULL if no data                        **
```

```
**    RETURN VALUE   :                                                    **
**       EM_SUCCESS or EM_ERROR                                           **
**                                                                        **
************************************************************************

************************************************************************
**                                                                        **
**    FUNCTION       : int CMSDelMsgHandler()                             **
**    DESCRIPTION    :                                                    **
**       Disassociates <Handler> with the reception of messages           **
**    of type <MsgType>.                                                  **
**                                                                        **
**    ARGUMENTS      :                                                    **
**       int            MsgType -- MUST BE POSITIVE                      **
**    RETURN VALUE   :                                                    **
**       EM_SUCCESS or S_NOT_FOUND.                                       **
**                                                                        **
************************************************************************

************************************************************************
**                                                                        **
**    FUNCTION       : int CMSProcessMsg()                                **
**    DESCRIPTION    :                                                    **
**       Formally processes a SINGLE incoming message. If no pending      **
**    message exists, the caller is put to sleep until                    **
**    such an event occurs. <IdRef> specifies the desired source          **
**    identity. CMS_ANY_IDENTITY may be used to denote any source         **
**    identity. <Type> specifies the type of messages desired.            **
**    CMS_ANY_MSG may be used to denote any message type. Upon            **
**    reception of a message, the associated message handler is           **
**    envoked.                                                            **
**    The caller can effectvely parse the message buffer by making        **
**    successive calls to CMSGetNextObjType() followed by the             **
**    appropriate "Get" function.                                         **
**       Note, the caller is responsible for calling CMSDestroyBuf()      **
**    when the message buffer is no longer required...                    **
**                                                                        **
**    ARGUMENTS      :                                                    **
**       CMS_IdentityRef   IdRef                                          **
**       int               Type                                           **
**    RETURN VALUE   :                                                    **
**       EM_SUCCESS or EM_ERROR.                                          **
**                                                                        **
************************************************************************

************************************************************************
**                                                                        **
**    FUNCTION       : int CMSProcessMsgs()                               **
**    DESCRIPTION    :                                                    **
**       Formally processes incoming messages. If no pending              **
**    messages exist, the caller is put to sleep until                    **
**    such an event occurs. <IdRef> specifies the desired source          **
**    identity. CMS_ANY_IDENTITY may be used to denote any source         **
**    identity. <Type> specifies the type of messages desired.            **
```

```
**    CMS_ANY_MSG may be used to denote any message type. Upon        **
**    reception of a message, the associated message handler is       **
**    envoked.                                                        **
**    The caller can effectvely parse the message buffer by making    **
**    successive calls to CMSGetNextObjType() followed by the         **
**    appropriate "Get" function.                                     **
**       Note, the caller is responsible for calling CMSDestroyBuf()  **
**    when the message buffer is no longer required...                **
**                                                                    **
**    ARGUMENTS      :                                                **
**       CMS_IdentityRef   IdRef                                      **
**       int               Type                                       **
**    RETURN VALUE   :                                                **
**       EM_SUCCESS or EM_ERROR.                                      **
**                                                                    **
************************************************************************

************************************************************************
**                                                                    **
**    FUNCTION       : int CMSQueryMsg()                              **
**    DESCRIPTION    :                                                **
**       Formally processes a SINGLE incoming message. If no pending  **
**    message exists, this call returns IMMEDIATELY with a return     **
**    code of CMS_NONE_PENDING. <IdRef> specifies the desired source  **
**    identity. CMS_ANY_IDENTITY may be used to denote any source     **
**    identity. <Type> specifies the type of messages desired.        **
**    CMS_ANY_MSG may be used to denote any message type. Upon        **
**    reception of a message, the associated message handler is       **
**    envoked.                                                        **
**    The caller can effectvely parse the message buffer by making    **
**    successive calls to CMSGetNextObjType() followed by the         **
**    appropriate "Get" function.                                     **
**       Note, the caller is responsible for calling CMSDestroyBuf()  **
**    when the message buffer is no longer required...                **
**                                                                    **
**    ARGUMENTS      :                                                **
**       CMS_IdentityRef   IdRef                                      **
**       int               Type                                       **
**    RETURN VALUE   :                                                **
**       EM_SUCCESS, CMS_NONE_PENDING, or EM_ERROR                    **
**                                                                    **
************************************************************************

************************************************************************
**                                                                    **
**    FUNCTION       : int CMSQueryMsgs()                             **
**    DESCRIPTION    :                                                **
**       Formally processes incoming messages. If no pending          **
**    messages exist, this call returns IMMEDIATELY with a return     **
**    code of EM_SUCCESS. <IdRef> specifies the desired source        **
**    identity. CMS_ANY_IDENTITY may be used to denote any source     **
**    identity. <Type> specifies the type of messages desired.        **
**    CMS_ANY_MSG may be used to denote any message type. Upon        **
**    reception of a message, the associated message handler is       **
```

```
**    envoked.                                                          **
**    The caller can effectvely parse the message buffer by making      **
**    successive calls to CMSGetNextObjType() followed by the           **
**    appropriate "Get" function.                                       **
**       Note, the caller is responsible for calling CMSDestroyBuf()    **
**    when the message buffer is no longer required...                  **
**                                                                      **
**    ARGUMENTS      :                                                  **
**       CMS_IdentityRef   IdRef                                        **
**       int               Type                                         **
**    RETURN VALUE   :                                                  **
**       EM_SUCCESS or EM_ERROR                                         **
**                                                                      **
**************************************************************************
```

BUFFER MANAGER:

```
**********************************************************************
**                                                                  **
**   FUNCTION       : CMS_BufRef CMSCreateBuf()                     **
**   DESCRIPTION    :                                               **
**      Creates and activates a "send" buffer which encodes its     **
**   contents according to <Encoding>. The specific type of encoding**
**   is persistant throughout the life of the buffer. This          **
**   buffer is intended to contain "CMS client (external)" objects. **
**                                                                  **
**   ARGUMENTS      :                                               **
**      CMS_Encoding      Encoding :                                **
**                          CMS_DATA_DEFAULT  - Encodes data for    **
**                                              heterogeneous networks**
**                          CMS_RAW       - No encoding takes place **
**                          CMS_IN_PLACE - Same as CMS_DATA_DEFAULT **
**                                         except that the data is  **
**                                         not copied into the      **
**                                         buffer until it is sent. **
**                                         Only references and      **
**                                         sizes are stored.        **
**                                                                  **
**   RETURN VALUE   :                                               **
**      A reference to a buffer or EM_ERROR_REF                     **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**   FUNCTION       : int CMSDestroyBuf()                           **
**   DESCRIPTION    :                                               **
**      Destroys the buffer referenced by <BufRef>.                 **
**                                                                  **
**   ARGUMENTS      :                                               **
**      CMS_BufRef         BufRef                                   **
**   RETURN VALUE   :                                               **
**      EM_SUCCESS or EM_ERROR                                      **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**   FUNCTION       : void CMSPrintBuf()                            **
**   DESCRIPTION    :                                               **
**      Prints information about <BufRef> to stdout.                **
**                                                                  **
**   ARGUMENTS      :                                               **
**      CMS_BufRef         BufRef                                   **
**   RETURN VALUE   :                                               **
**                                                                  **
**********************************************************************
```

OBJECT MANAGER:

```
**********************************************************************
**                                                                  **
**   FUNCTION       : CMS_InstanceIdRef CMSCreateType1InstanceId()  **
**   DESCRIPTION    :                                               **
**      Create an instance Id of type1.                             **
**                                                                  **
**   ARGUMENTS      :                                               **
**      long              LVal                                      **
**                                                                  **
**   RETURN VALUE   :                                               **
**      A reference to an instance Id or EM_ERROR_REF               **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**   FUNCTION       : CMS_InstanceIdRef CMSCreateType2InstanceId()  **
**   DESCRIPTION    :                                               **
**      Create an instance Id of type2.                             **
**                                                                  **
**   ARGUMENTS      :                                               **
**      long              LVal                                      **
**      char              *SVal                                     **
**                                                                  **
**   RETURN VALUE   :                                               **
**      A reference to an instance Id or EM_ERROR_REF               **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**   FUNCTION       : CMS_InstanceIdRef CMSCreateType3InstanceId()  **
**   DESCRIPTION    :                                               **
**      Create an instance Id of type3.                             **
**                                                                  **
**   ARGUMENTS      :                                               **
**      long              LVal                                      **
**      char              *SVal1                                    **
**      char              *SVal2                                    **
**                                                                  **
**   RETURN VALUE   :                                               **
**      A reference to an instance Id or EM_ERROR_REF               **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**   FUNCTION       : CMS_InstanceIdRef CMSCreateType4InstanceId()  **
**   DESCRIPTION    :                                               **
**      Create an instance Id of type4.                             **
**                                                                  **
**   ARGUMENTS      :                                               **
```

```
**      long              LVal                                          **
**      char              *SVal1                                        **
**      char              *SVal2                                        **
**      char              *SVal3                                        **
**                                                                      **
**   RETURN VALUE   :                                                   **
**      A reference to an instance Id or EM_ERROR_REF                   **
**                                                                      **
**************************************************************************

**************************************************************************
**                                                                      **
**   FUNCTION       : void CMSDestroyInstanceId()                       **
**   DESCRIPTION    :                                                   **
**      Destroys <InstanceIdRef>.                                       **
**                                                                      **
**   ARGUMENTS      :                                                   **
**      CMS_InstanceIdRef InstanceIdRef                                 **
**                                                                      **
**   RETURN VALUE   :                                                   **
**                                                                      **
**************************************************************************

**************************************************************************
**                                                                      **
**   FUNCTION       : CMS_SlotRef CMSCreateSlot()                       **
**   DESCRIPTION    :                                                   **
**      Create a slot object.                                           **
**                                                                      **
**   ARGUMENTS      :                                                   **
**      int               SlotId                                        **
**      CMS_SlotType      Type                                          **
**      CMS_SlotValue     Value                                         **
**                                                                      **
**   RETURN VALUE   :                                                   **
**      A reference to a slot or EM_ERROR_REF                           **
**                                                                      **
**************************************************************************

**************************************************************************
**                                                                      **
**   FUNCTION       : void CMSDestroySlot()                             **
**   DESCRIPTION    :                                                   **
**      Destroys <SlotRef>.                                             **
**                                                                      **
**   ARGUMENTS      :                                                   **
**      CMS_SlotRef       SlotRef                                       **
**                                                                      **
**   RETURN VALUE   :                                                   **
**                                                                      **
**************************************************************************

**************************************************************************
**                                                                      **
```

```
**    FUNCTION       : CMS_SlotListRef CMSCreateSlotList()                **
**    DESCRIPTION    :                                                    **
**       Create an empty slot list representing an object with           **
**    class Id <ClassId> and instance Id <InstanceIdRef>. Object          **
**    slots can be added to the list via calls to CMSAddSlot().           **
**                                                                         **
**    ARGUMENTS      :                                                    **
**       int                 ClassId                                     **
**       CMS_InstanceIdRef InstanceIdRef                                 **
**                                                                        **
**    RETURN VALUE   :                                                    **
**       A reference to a slot list or EM_ERROR_REF                       **
**                                                                        **
*****************************************************************************

*****************************************************************************
**                                                                         **
**    FUNCTION       : void CMSDestroySlotList()                          **
**    DESCRIPTION    :                                                    **
**       Destroys <ListRef>. Note, this function also destroys the        **
**    associated instance id.                                             **
**                                                                        **
**    ARGUMENTS      :                                                    **
**       CMS_SlotListRef   ListRef                                        **
**                                                                        **
**    RETURN VALUE   :                                                    **
**       EM_SUCCESS or EM_ERROR                                           **
**                                                                        **
*****************************************************************************

*****************************************************************************
**                                                                         **
**    FUNCTION       : CMS_MERef CMSCreateME()                            **
**    DESCRIPTION    :                                                    **
**       Creates a CMS_ME object.                                         **
**                                                                        **
**    ARGUMENTS      :                                                    **
**       CMS_METype          Type                                        **
**       S_VoidRef           ValRef                                      **
**                                                                        **
**    RETURN VALUE   :                                                    **
**       A reference to a CMS_ME object or EM_ERROR_REF.                  **
**                                                                        **
*****************************************************************************

*****************************************************************************
**                                                                         **
**    FUNCTION       : void CMSDestroyME()                                **
**    DESCRIPTION    :                                                    **
**       Destroys <MERef>.                                                **
**                                                                        **
**    ARGUMENTS      :                                                    **
**       CMS_MERef           MERef                                       **
**                                                                        **
```

```
**    RETURN VALUE   :                                                    **
**                                                                        **
************************************************************************

************************************************************************
**                                                                      **
**    FUNCTION       : void CMSDestroyMEValList()                       **
**    DESCRIPTION    :                                                  **
**       Destroys the list headed by <MEHeadRef>.                       **
**                                                                      **
**    ARGUMENTS      :                                                  **
**       CMS_MERef          MEHeadRef                                   **
**                                                                      **
**    RETURN VALUE   :                                                  **
**                                                                      **
************************************************************************

************************************************************************
**                                                                      **
**    FUNCTION       : CMS_MERef CMSAddME()                             **
**    DESCRIPTION    :                                                  **
**       Adds <MERef> to the end of the list of "multi-elements"        **
**    of which <MEHeadRef> is the first element of the list. If         **
**    no elements exist in the list (ie. this is the first call)        **
**    a value of CMS_NEW_ME_LIST should be used for <MEHeadRef>.        **
**    Note, <MERef> is NOT copied into the list. Rather the list        **
**    acquires ownership of <MERef>.                                    **
**                                                                      **
**    ARGUMENTS      :                                                  **
**       CMS_MERef          MEHeadRef                                   **
**       CMS_MERef          MERef                                       **
**                                                                      **
**    RETURN VALUE   :                                                  **
**       A reference to the list or EM_ERROR_REF.                       **
**                                                                      **
************************************************************************

************************************************************************
**                                                                      **
**    FUNCTION       : CMS_MERef CMSGetNextME()                         **
**    DESCRIPTION    :                                                  **
**       Returns a reference to the next CMS_ME object in the           **
**    "multi-element" list associated with <MERef>. To parse a          **
**    "multi-element" slot, obtain the ME list via a call to            **
**    _CMSGetMEValList(). Then each call to CMSGetNextME() will         **
**    return the next CMS_ME object in the list with respect to         **
**    <MERef>. When the end of the list is reached, NULL is returned.   **
**                                                                      **
**    ARGUMENTS      :                                                  **
**       CMS_MERef          MERef                                       **
**                                                                      **
**    RETURN VALUE   :                                                  **
**       A reference to a CMS_ME object or NULL                         **
**                                                                      **
```

```
*************************************************************************

*************************************************************************
**                                                                     **
**    FUNCTION       : void CMSAddSlot()                               **
**    DESCRIPTION    :                                                 **
**       Adds <SlotRef> to <ListRef>.                                  **
**                                                                     **
**    ARGUMENTS      :                                                 **
**       CMS_SlotListRef   ListRef                                     **
**       CMS_SlotRef       SlotRef                                     **
**                                                                     **
**    RETURN VALUE   :                                                 **
**       EM_SUCCESS or EM_ERROR                                        **
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**    FUNCTION       : void CMSApplyToSlotList()                       **
**    DESCRIPTION    :                                                 **
**       Applies <Func> to all slots in <ListRef> passing <Data>       **
**    to <Func> each time it is called. <Data> allows clients          **
**    to pass client defined data to <Func>. The following             **
**    arguments are passed to <Func> for each slot in <ListRef>:       **
**                                                                     **
**       int               ClassId                                     **
**       CMS_InstanceIdRef InstanceIdRef                               **
**       CMS_SlotRef       CurSlotRef                                  **
**       S_VoidRef         DataRef                                     **
**                                                                     **
**                                                                     **
**    ARGUMENTS      :                                                 **
**       CMS_SlotListRef   ListRef                                     **
**       CMS_VFunc         Func                                        **
**       S_VoidRef         DataRef                                     **
**                                                                     **
**    RETURN VALUE   :                                                 **
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**    FUNCTION       : int CMSPutCreateObj()                           **
**    DESCRIPTION    :                                                 **
**       Copies the object referenced by <ListRef> into the buffer     **
**    specified by <BufRef> with an object type of CMS_CREATE_OBJ.     **
**                                                                     **
**    ARGUMENTS      :                                                 **
**       CMS_BufRef        BufRef                                      **
**       CSM_SlotList      ListRef                                     **
**                                                                     **
**    RETURN VALUE   :                                                 **
**       EM_SUCCESS or EM_ERROR                                        **
```

```
**                                                                    **
**********************************************************************

**********************************************************************
**                                                                    **
**   FUNCTION       : int CMSPutUpdateObj()                          **
**   DESCRIPTION    :                                                **
**      Copies the object referenced by <ListRef> into the buffer     **
**   specified by <BufRef> with an object type of CMS_UPDATE_OBJ.     **
**                                                                    **
**   ARGUMENTS      :                                                **
**      CMS_BufRef          BufRef                                    **
**      CSM_SlotList        ListRef                                   **
**                                                                    **
**   RETURN VALUE   :                                                **
**      EM_SUCCESS or EM_ERROR                                        **
**                                                                    **
**********************************************************************

**********************************************************************
**                                                                    **
**   FUNCTION       : int CMSPutDeleteObj()                          **
**   DESCRIPTION    :                                                **
**      Copies the object referenced by <ListRef> into the buffer     **
**   specified by <BufRef> with an object type of CMS_DELETE_OBJ.     **
**                                                                    **
**   ARGUMENTS      :                                                **
**      CMS_BufRef          BufRef                                    **
**      CSM_SlotList        ListRef                                   **
**                                                                    **
**   RETURN VALUE   :                                                **
**      EM_SUCCESS or EM_ERROR                                        **
**                                                                    **
**********************************************************************

**********************************************************************
**                                                                    **
**   FUNCTION       : CSM_SlotListRef CMSGetObj()                    **
**   DESCRIPTION    :                                                **
**      Removes and returns the next object in <BufRef> as a          **
**   CMS_SlotList.                                                    **
**                                                                    **
**   ARGUMENTS      :                                                **
**      CMS_BufRef          BufRef                                    **
**                                                                    **
**   RETURN VALUE   :                                                **
**      A reference to a slot list or EM_ERROR_REF.                   **
**                                                                    **
**********************************************************************

**********************************************************************
**                                                                    **
**   FUNCTION       : void CMSPrintSlot()                            **
**   DESCRIPTION    :                                                **
```

```
**      Prints <SlotRef> to stdout.                                    **
**                                                                     **
**   ARGUMENTS      :                                                  **
**      CMS_SlotRef        SlotRef                                     **
**                                                                     **
**   RETURN VALUE   :                                                  **
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**   FUNCTION       : int CMSPrintSlotList()                           **
**   DESCRIPTION    :                                                  **
**      Prints <SlotListRef> to stdout.                                **
**                                                                     **
**   ARGUMENTS      :                                                  **
**      CMS_SlotListRef    ListRef                                     **
**                                                                     **
**   RETURN VALUE   :                                                  **
**      EM_SUCCESS or EM_ERROR                                         **
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**   FUNCTION       : int CMSCreateRmtIdentity()                       **
**   DESCRIPTION    :                                                  **
**      Creates a remote copy of <IdRef> in <TrgtIdRef>.               **
**                                                                     **
**   ARGUMENTS      :                                                  **
**      CMS_IdentityRef    TrgtIdRef                                   **
**      int                MsgType -- MUST BE POSITIVE                 **
**      CMS_IdentityRef    IdRef                                       **
**                                                                     **
**   RETURN VALUE   :                                                  **
**      EM_SUCCESS or EM_ERROR                                         **
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**   FUNCTION       : int CMSGrpCreateRmtIdentity()                    **
**   DESCRIPTION    :                                                  **
**      Creates a remote copy of <IdRef> in all members of <GrpName>   **
**                                                                     **
**   ARGUMENTS      :                                                  **
**      char               *GrpNamef                                   **
**      int                MsgType -- MUST BE POSITIVE                 **
**      CMS_IdentityRef    IdRef                                       **
**                                                                     **
**   RETURN VALUE   :                                                  **
**      EM_SUCCESS or EM_ERROR                                         **
**                                                                     **
*************************************************************************
```

```
**********************************************************************
**                                                                  **
**    FUNCTION       : int CMSCreateRmtObj()                        **
**    DESCRIPTION    :                                              **
**       Creates a remote object represented by <ObjRef> in         **
**    <TrgtIdRef>.                                                  **
**                                                                  **
**    ARGUMENTS      :                                              **
**       CMS_IdentityRef   TrgtIdRef                                **
**       int               MsgType -- MUST BE POSITIVE              **
**       CMS_SlotListRef   ObjRef                                   **
**                                                                  **
**    RETURN VALUE   :                                              **
**       EM_SUCCESS or EM_ERROR                                     **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**    FUNCTION       : int CMSGrpCreateRmtObj()                     **
**    DESCRIPTION    :                                              **
**       Creates a remote object represented by <ObjRef> in all     **
**    members of <GrpName>.                                         **
**                                                                  **
**    ARGUMENTS      :                                              **
**       char              GrpName                                  **
**       int               MsgType -- MUST BE POSITIVE              **
**       CMS_SlotListRef   ObjRef                                   **
**                                                                  **
**    RETURN VALUE   :                                              **
**       EM_SUCCESS or EM_ERROR                                     **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**    FUNCTION       : int CMSUpdateRmtObj()                        **
**    DESCRIPTION    :                                              **
**       Updates a remote object represented by <ObjRef> in         **
**    <TrgtIdRef>.                                                  **
**                                                                  **
**    ARGUMENTS      :                                              **
**       CMS_IdentityRef   TrgtIdRef                                **
**       int               MsgType -- MUST BE POSITIVE              **
**       CMS_SlotListRef   ObjRef                                   **
**                                                                  **
**    RETURN VALUE   :                                              **
**       EM_SUCCESS or EM_ERROR                                     **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**    FUNCTION       : int CMSGrpUpdateRmtObj()                     **
```

```
**    DESCRIPTION    :                                                    **
**       Updates a remote object represented by <ObjRef> in all          **
**    members of <GrpName>.                                               **
**                                                                        **
**    ARGUMENTS      :                                                    **
**       char               GrpName                                      **
**       int                MsgType -- MUST BE POSITIVE                  **
**       CMS_SlotListRef    ObjRef                                       **
**                                                                        **
**    RETURN VALUE   :                                                    **
**       EM_SUCCESS or EM_ERROR                                           **
**                                                                        **
************************************************************************

************************************************************************
**                                                                        **
**    FUNCTION       : int CMSDeleteRmtObj()                           **
**    DESCRIPTION    :                                                    **
**       Deletes a remote object represented by <ObjRef> in            **
**    <TrgtIdRef>.                                                        **
**                                                                        **
**    ARGUMENTS      :                                                    **
**       CMS_IdentityRef    TrgtIdRef                                    **
**       int                MsgType -- MUST BE POSITIVE                  **
**       CMS_SlotListRef    ObjRef                                       **
**                                                                        **
**    RETURN VALUE   :                                                    **
**       EM_SUCCESS or EM_ERROR                                           **
**                                                                        **
************************************************************************

************************************************************************
**                                                                        **
**    FUNCTION       : int CMSGrpDeleteRmtObj()                        **
**    DESCRIPTION    :                                                    **
**       Deletes a remote object represented by <ObjRef> in all          **
**    members of <GrpName>.                                               **
**                                                                        **
**    ARGUMENTS      :                                                    **
**       char               GrpName                                      **
**       int                MsgType -- MUST BE POSITIVE                  **
**       CMS_SlotListRef    ObjRef                                       **
**                                                                        **
**    RETURN VALUE   :                                                    **
**       EM_SUCCESS or EM_ERROR                                           **
**                                                                        **
************************************************************************

************************************************************************
**                                                                        **
**    FUNCTION       : int CMSSendBuf()                                **
**    DESCRIPTION    :                                                    **
**       Sends the contents of <BufRef> to <TrgtIdRef>.                   **
**                                                                        **
```

```
**    ARGUMENTS      :                                                    **
**       CMS_IdentityRef   TrgtIdRef                                     **
**       int               MsgType -- MUST BE POSITIVE                   **
**       CMS_BufRef        BufRef                                        **
**                                                                       **
**    RETURN VALUE   :                                                   **
**       EM_SUCCESS or EM_ERROR                                          **
**                                                                       **
***************************************************************************

***************************************************************************
**                                                                       **
**    FUNCTION       : int CMSGrpSendBuf()                               **
**    DESCRIPTION    :                                                   **
**       Sends the contents of <BufRef> to all members of <GrpName>.     **
**                                                                       **
**    ARGUMENTS      :                                                   **
**       char              *GrpName                                      **
**       int               MsgType -- MUST BE POSITIVE                   **
**       CMS_BufRef        BufRef                                        **
**                                                                       **
**    RETURN VALUE   :                                                   **
**       EM_SUCCESS or EM_ERROR                                          **
**                                                                       **
***************************************************************************

***************************************************************************
**                                                                       **
**    FUNCTION       : CMS_ObjType CMSGetNextObjType()                   **
**    DESCRIPTION    :                                                   **
**       Obtains the type of the next object in                          **
**    <BufRef>. Note, this function can only be called once for          **
**    each object in <BufRef>. When all objects contained in             **
**    <BufRef> have been read, a value of CMS_END_OF_BUF is returned.    **
**       This function should be used by clients as a means of           **
**    parsing a newly received message buffer. Clients should            **
**    call the appropriate "Get" function based on this value.           **
**                                                                       **
**    ARGUMENTS      :                                                   **
**       CMS_BufRef        BufRef                                        **
**                                                                       **
**    RETURN VALUE   :                                                   **
**       An object type, CMS_END_OF_BUF, or EM_ERROR.                    **
**                                                                       **
***************************************************************************
```

Part (2): CLIPS Object-Based Functional Interface

CLIPS USER FUNCTIONS:

```
*****************************************************************
**                                                             **
**   FUNCTION       : int CMSConnect                           **
**   DESCRIPTION    :                                          **
**      Connects the caller to a multi-agent session.          **
**                                                             **
**   ARGUMENTS      :                                          **
**      SYMBOL_OR_STRING  Name                                 **
**      SYMBOL_OR_STRING  ClassDescFileName - (optional)       **
**                                            Specifies the name **
**                                            of the file which  **
**                                            describes the classes **
**                                            which are candidates  **
**                                            to be communicated.   **
**                                            If no file name is    **
**                                            specified, ~icodes2/  **
**                                            run/config/           **
**                                            ClassTable.dat is used**
**   RETURN VALUE   :                                          **
**      A reference to the object representation of the caller **
**   or EM_ERROR_REF. Note, this object can be passed to other **
**   clients as a way to address the caller directly...        **
**                                                             **
*****************************************************************

*****************************************************************
**                                                             **
**   FUNCTION       : int CMSDisconnect                        **
**   DESCRIPTION    :                                          **
**      Disconnects the caller from a multi-agent session.     **
**                                                             **
**   ARGUMENTS      :                                          **
**   RETURN VALUE   :                                          **
**      EM_SUCCESS or EM_ERROR                                 **
**                                                             **
*****************************************************************

*****************************************************************
**                                                             **
**   FUNCTION       : int CMSEnterGrp                          **
**   DESCRIPTION    :                                          **
**      Enrolls the caller into the group <Group>.             **
**                                                             **
**   ARGUMENTS      :                                          **
**      SYMBOL_OR_STRING  Group                                **
**   RETURN VALUE   :                                          **
**      The caller's instance number within <Group> or EM_ERROR.  **
**   Instance numbers start at 0 and increment upward.         **
**                                                             **
*****************************************************************
```

```
**********************************************************************
**                                                                  **
**   FUNCTION       : int CMSExitGrp                                **
**   DESCRIPTION    :                                               **
**      Unenrolls the caller from the group <Group>.                **
**                                                                  **
**   ARGUMENTS      :                                               **
**      SYMBOL_OR_STRING  Group                                     **
**   RETURN VALUE   :                                               **
**      EM_SUCCESS or EM_ERROR                                      **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**   FUNCTION       : int CMSSpawn                                  **
**   DESCRIPTION    :                                               **
**      Spawns <NumCopies> copies of <Task> in accordance with      **
**   the values of <How> and <where>.                               **
**                                                                  **
**   ARGUMENTS      :                                               **
**      SYMBOL            Task                                      **
**      INTEGER           NumCopies                                 **
**      INTEGER           How :                                     **
**                           CMS_TASK_DEFAULT - Target host is      **
**                                              determined by the   **
**                                              Session Manager. In **
**                                              this case <Where> should **
**                                              be NULL.            **
**                           CMS_HOST       - Target host is        **
**                                            determined by <Where>. **
**                           CMS_ARCH       - Target host is        **
**                                            determined by the     **
**                                            Session Manager. The  **
**                                            target host's architecture**
**                                            will be of type <Where>. **
**                           CMS_DEBUG      - Target host is        **
**                                            determined by the     **
**                                            Session Manager. In   **
**                                            this case <Where> should **
**                                            be NULL. All copies of **
**                                            <Task> will be run in **
**                                            the PVM debugger...   **
**                           CMS_TRACE      - Target host is        **
**                                            determined by the     **
**                                            Session Manager. In   **
**                                            this case <Where> should **
**                                            be NULL. All copies of **
**                                            <Task> will generate PVM **
**                                            trace data...         **
**      SYMBOL            Where           - Examples :            **
**                                          "moby.cadrc.calpoly.edu" **
**                                          or "HPPA"               **
**      ANY # OF ARGS.    ArgV            - Arguments passed to     **
```

```
**                                          each task upon startup.    **
**                                          Note, Argv may be a        **
**                                          collection of any number   **
**                                          of BASIC data type atoms   **
**                                          (ie. SYMBOL, STRING,       **
**                                          INTEGER, etc).             **
**                                                                     **
**    RETURN VALUE  :                                                  **
**       The actual number of successfully spawned tasks or EM_ERROR.**
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**    FUNCTION      : int CMSPrintIdentity                             **
**    DESCRIPTION   :                                                  **
**       Prints the object referenced by <IdentityRef> to stdout.      **
**                                                                     **
**    ARGUMENTS     :                                                  **
**       CMS_IdentityRef   IdentityRef                                 **
**    RETURN VALUE  :                                                  **
**       EM_SUCCESS or EM_ERROR                                        **
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**    FUNCTION      : CMS_BufRef CMSCreateBuf                          **
**    DESCRIPTION   :                                                  **
**       Creates a "send" buffer which encodes its contents            **
**    according to <Encoding>. The specific type of encoding           **
**    is persistant throughout the life of the buffer...               **
**                                                                     **
**    ARGUMENTS     :                                                  **
**       INTEGER           Encoding :                                  **
**                            CMS_DATA_DEFAULT  - Encodes data for     **
**                                                heterogeneous networks**
**                            CMS_RAW         - No encoding takes place**
**                            CMS_IN_PLACE - Same as CMS_DEFAULT       **
**                                           except that the data is   **
**                                           not copied into the       **
**                                           buffer until it is sent.  **
**                                           Only sizes and references**
**                                           are stored.               **
**                                                                     **
**    RETURN VALUE  :                                                  **
**       A reference to a buffer or EM_ERROR_REF                       **
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**    FUNCTION      : int CMSDestroyBuf                                **
**    DESCRIPTION   :                                                  **
```

```
**      Destroys the buffer referenced by <BufRef>.                    **
**                                                                     **
**   ARGUMENTS     :                                                   **
**      CMS_BufRef        BufRef                                       **
**   RETURN VALUE  :                                                   **
**      EM_SUCCESS or EM_ERROR                                         **
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**   FUNCTION      : int CMSProcessMsg                                 **
**   DESCRIPTION   :                                                   **
**      Formally processes a SINGLE incoming message. If no pending    **
**   messages exist, the caller is put to sleep until                  **
**   such an event occurs. <IdRef> specifies the desired source        **
**   identity. CMS_ANY_IDENTITY may be used to denote any source       **
**   identity. <Type> specifies the type of messages desired.          **
**   CMS_ANY_MSG may be used to denote any message type. Upon          **
**   reception of a message, the contents are processed as follows:    **
**      - Identities are asserted into the callers fact list as        **
**           SESSION-MEMBER <Name> CAN-BE-REFERENCED-BY <IdRef>        **
**      - Objects are created, updated, or deleted accordingly         **
**                                                                     **
**   ARGUMENTS     :                                                   **
**      CMS_IdentityRef   IdRef                                        **
**      int               MsgType                                      **
**   RETURN VALUE  :                                                   **
**      EM_SUCCESS or EM_ERROR                                         **
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**   FUNCTION      : int CMSProcessMsgs                                **
**   DESCRIPTION   :                                                   **
**      Formally processes incoming messages. If no pending            **
**   messages exist, the caller is put to sleep until                  **
**   such an event occurs. <IdRef> specifies the desired source        **
**   identity. CMS_ANY_IDENTITY may be used to denote any source       **
**   identity. <Type> specifies the type of messages desired.          **
**   CMS_ANY_MSG may be used to denote any message type. Upon          **
**   reception of a message, the contents are processed as follows:    **
**      - Identities are asserted into the callers fact list as        **
**           SESSION-MEMBER <Name> CAN-BE-REFERENCED-BY <IdRef>        **
**      - Objects are created, updated, or deleted accordingly         **
**                                                                     **
**   ARGUMENTS     :                                                   **
**      CMS_IdentityRef   IdRef                                        **
**      int               MsgType                                      **
**   RETURN VALUE  :                                                   **
**      EM_SUCCESS or EM_ERROR                                         **
**                                                                     **
*************************************************************************
```

```
*****************************************************************
**                                                             **
**   FUNCTION       : int CMSQueryMsg                          **
**   DESCRIPTION    :                                          **
**      Formally processes a SINGLE incoming message. If no pending **
**   messages exist, this call returns IMMEDIATELY with a return  **
**   code of CMS_NONE_PENDING. <IdRef> specifies the desired source **
**   identity. CMS_ANY_IDENTITY may be used to denote any source   **
**   identity. <Type> specifies the type of messages desired.      **
**   CMS_ANY_MSG may be used to denote any message type. Upon      **
**   reception of a message, the contents are processed as follows: **
**      - Identities are asserted into the callers fact list as    **
**           <Name> CAN-BE-REFERENCED-BY <IdRef>                   **
**      - Objects are created, updated, or deleted accordingly     **
**                                                                 **
**   ARGUMENTS      :                                           **
**      CMS_IdentityRef   IdRef                                    **
**      int               MsgType                                  **
**   RETURN VALUE   :                                           **
**      EM_SUCCESS, CMS_NONE_PENDING, or EM_ERROR                  **
**                                                                 **
*****************************************************************

*****************************************************************
**                                                             **
**   FUNCTION       : int CMSQueryMsgs                         **
**   DESCRIPTION    :                                          **
**      Formally processes incoming messages. If no pending        **
**   messages exist, this call returns IMMEDIATELY with a return   **
**   code of EM_SUCCESS. <IdRef> specifies the desired source      **
**   identity. CMS_ANY_IDENTITY may be used to denote any source   **
**   identity. <Type> specifies the type of messages desired.      **
**   CMS_ANY_MSG may be used to denote any message type. Upon      **
**   reception of a message, the contents are processed as follows: **
**      - Identities are asserted into the callers fact list as    **
**           <Name> CAN-BE-REFERENCED-BY <IdRef>                   **
**      - Objects are created, updated, or deleted accordingly     **
**                                                                 **
**   ARGUMENTS      :                                           **
**      CMS_IdentityRef   IdRef                                    **
**      int               MsgType                                  **
**   RETURN VALUE   :                                           **
**      EM_SUCCESS or EM_ERROR                                     **
**                                                                 **
*****************************************************************

*****************************************************************
**                                                             **
**   FUNCTION       : INSTANCE_NAME CMSBuildInstanceName       **
**   DESCRIPTION    :                                          **
**      Builds a valid instance name based on <ClassName> and the  **
**   instance Id.                                                  **
**                                                                 **
**   ARGUMENTS      :                                           **
```

```
**      SYMBOL_OR_STRING  ClassName                                    **
**                                                                     **
**             EITHER                                                  **
**      LONG              LVal                                         **
**               OR                                                    **
**      LONG              LVal                                         **
**      SYMBOL_OR_STRING  S1Val                                        **
**               OR                                                    **
**      LONG              LVal                                         **
**      SYMBOL_OR_STRING  S1Val                                        **
**      SYMBOL_OR_STRING  S2Val                                        **
**               OR                                                    **
**      LONG              LVal                                         **
**      SYMBOL_OR_STRING  S1Val                                        **
**      SYMBOL_OR_STRING  S2Val                                        **
**      SYMBOL_OR_STRING  S3Val                                        **
**                                                                     **
**   RETURN VALUE  :                                                   **
**      An INSTANCE_NAME or NULL                                       **
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**   FUNCTION      : int CMSPutCreateObj                               **
**   DESCRIPTION   :                                                   **
**      Puts an object into <BufRef> with an object type of            **
**   CMS_CREATE_OBJ.                                                   **
**                                                                     **
**   ARGUMENTS     :                                                   **
**      CMS_BufRef        BufRef                                       **
**      INSTANCE_NAME     InstName                                     **
**      Any number of slot overrides...                                **
**   RETURN VALUE  :                                                   **
**      EM_SUCCESS or EM_ERROR                                         **
**                                                                     **
*************************************************************************

*************************************************************************
**                                                                     **
**   FUNCTION      : int CMSPutUpdateObj                               **
**   DESCRIPTION   :                                                   **
**      Puts an object into <BufRef> with an object type of            **
**   CMS_UPDATE_OBJ.                                                   **
**                                                                     **
**   ARGUMENTS     :                                                   **
**      CMS_BufRef        BufRef                                       **
**      INSTANCE_NAME     InstName                                     **
**      Any number of slot overrides...                                **
**   RETURN VALUE  :                                                   **
**      EM_SUCCESS or EM_ERROR                                         **
**                                                                     **
*************************************************************************
```

```
**********************************************************************
**                                                                  **
**   FUNCTION       : int CMSPutDeleteObj                           **
**   DESCRIPTION    :                                               **
**      Puts an object into <BufRef> with an object type of         **
**   CMS_DELETE_OBJ.                                                **
**                                                                  **
**   ARGUMENTS      :                                               **
**      CMS_BufRef          BufRef                                  **
**      INSTANCE_NAME       InstName                                **
**      Any number of slot overrides...                             **
**   RETURN VALUE   :                                               **
**      EM_SUCCESS or EM_ERROR                                      **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**   FUNCTION       : int CMSCreateRmtIdentity                      **
**   DESCRIPTION    :                                               **
**      Creates a remote copy of <IdRef> in <TrgtIdRef>.            **
**                                                                  **
**   ARGUMENTS      :                                               **
**      CMS_IdentityRef   DstIdentityRef                            **
**      int               MsgType -- MUST BE POSITIVE               **
**      CMS_IdentityRef   ObjRef                                    **
**   RETURN VALUE   :                                               **
**      EM_SUCCESS or EM_ERROR                                      **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**   FUNCTION       : int CMSGrpCreateRmtIdentity                   **
**   DESCRIPTION    :                                               **
**      Creates a remote copy of <IdRef> in all members of <GrpName>**
**                                                                  **
**   ARGUMENTS      :                                               **
**      SYMBOL_OR_STRING  GrpName                                   **
**      int               MsgType -- MUST BE POSITIVE               **
**      CMS_IdentityRef   ObjRef                                    **
**   RETURN VALUE   :                                               **
**      EM_SUCCESS or EM_ERROR                                      **
**                                                                  **
**********************************************************************

**********************************************************************
**                                                                  **
**   FUNCTION       : int CMSCreateRmtObj                           **
**   DESCRIPTION    :                                               **
**      Creates a remote object in <DstIdRef>.                      **
**                                                                  **
**   ARGUMENTS      :                                               **
**      CMS_IdentityRef   DstIdRef                                  **
```

```
**      int                 MsgType -- MUST BE POSITIVE                 **
**      INSTANCE_NAME       InstName                                    **
**      Any number of slot overrides...                                 **
**   RETURN VALUE   :                                                   **
**      EM_SUCCESS or EM_ERROR                                          **
**                                                                      **
**************************************************************************

**************************************************************************
**                                                                      **
**   FUNCTION       : int CMSGrpCreateRmtObj                            **
**   DESCRIPTION    :                                                   **
**      Creates a remote object in all members of <GrpName>.            **
**                                                                      **
**   ARGUMENTS      :                                                   **
**      SYMBOL_OR_STRING  GrpName                                       **
**      int               MsgType -- MUST BE POSITIVE                   **
**      INSTANCE_NAME     InstName                                      **
**      Any number of slot overrides...                                 **
**   RETURN VALUE   :                                                   **
**      EM_SUCCESS or EM_ERROR                                          **
**                                                                      **
**************************************************************************

**************************************************************************
**                                                                      **
**   FUNCTION       : int CMSUpdateRmtObj                               **
**   DESCRIPTION    :                                                   **
**      Updates a remote object in <DstIdRef>.                          **
**                                                                      **
**   ARGUMENTS      :                                                   **
**      CMS_IdentityRef   DstIdRef                                      **
**      int               MsgType -- MUST BE POSITIVE                   **
**      INSTANCE_NAME     InstName                                      **
**      Any number of slot overrides...                                 **
**   RETURN VALUE   :                                                   **
**      EM_SUCCESS or EM_ERROR                                          **
**                                                                      **
**************************************************************************

**************************************************************************
**                                                                      **
**   FUNCTION       : int CMSGrpUpdateRmtObj                            **
**   DESCRIPTION    :                                                   **
**      Updates a remote object in all members of <GrpName>.            **
**                                                                      **
**   ARGUMENTS      :                                                   **
**      SYMBOL_OR_STRING  GrpName                                       **
**      int               MsgType -- MUST BE POSITIVE                   **
**      INSTANCE_NAME     InstName                                      **
**      Any number of slot overrides...                                 **
**   RETURN VALUE   :                                                   **
**      EM_SUCCESS or EM_ERROR                                          **
**                                                                      **
```

```
*****************************************************************

*****************************************************************
**                                                             **
**    FUNCTION       : int CMSDeleteRmtObj                     **
**    DESCRIPTION    :                                         **
**       Deletes a remote object in <DstIdRef>.                 **
**                                                             **
**    ARGUMENTS      :                                         **
**       CMS_IdentityRef   DstIdRef                            **
**       int               MsgType -- MUST BE POSITIVE         **
**       INSTANCE_NAME     InstName                            **
**       Any number of slot overrides...                       **
**    RETURN VALUE   :                                         **
**       EM_SUCCESS or EM_ERROR                                **
**                                                             **
*****************************************************************

*****************************************************************
**                                                             **
**    FUNCTION       : int CMSGrpDeleteRmtObj                  **
**    DESCRIPTION    :                                         **
**       Deletes a remote object in all members of <GrpName>.  **
**                                                             **
**    ARGUMENTS      :                                         **
**       SYMBOL_OR_STRING  GrpName                             **
**       int               MsgType -- MUST BE POSITIVE         **
**       INSTANCE_NAME     InstName                            **
**       Any number of slot overrides...                       **
**    RETURN VALUE   :                                         **
**       EM_SUCCESS or EM_ERROR                                **
**                                                             **
*****************************************************************

*****************************************************************
**                                                             **
**    FUNCTION       : int CMSSendBuf                          **
**    DESCRIPTION    :                                         **
**       Sends <BufRef> to <TrgtIdRef>                         **
**                                                             **
**    ARGUMENTS      :                                         **
**       CMS_IdentityRef   TrgtIdRef                           **
**       int               MsgType -- MUST BE POSITIVE         **
**       CMS_BufRef        BufRef                              **
**    RETURN VALUE   :                                         **
**       EM_SUCCESS or EM_ERROR                                **
**                                                             **
*****************************************************************

*****************************************************************
**                                                             **
**    FUNCTION       : int CMSGrpSendBuf                       **
**    DESCRIPTION    :                                         **
**       Sends <BufRef> to all members of <GrpName>.           **
```

```
**                                                                    **
**   ARGUMENTS      :                                                 **
**      SYMBOL_OR_STRING  GrpName                                     **
**      int               MsgType -- MUST BE POSITIVE                 **
**      CMS_BufRef        BufRef                                      **
**   RETURN VALUE   :                                                 **
**      EM_SUCCESS or EM_ERROR                                        **
**                                                                    **
************************************************************************
```

CAD Research Center - Technical Reports (Price List)

The following technical reports and conference proceedings are available from the CAD Research Center, College of Architecture and Environmental Design (CAED), Cal Poly, San Luis Obispo, CA 93407, USA. Prices include postage (surface mail for overseas orders - add 10% for overseas airmail).

CADRU-01-88 ($ 7.00)	'ICADS: Toward an Intelligent Computer-Aided Design System' (Pohl J., A.Chapman, L.Chirica and L.Myers, 1988)
CADRU-02-88 ($12.00)	'Implementation Strategies for a Prototype ICADS Working Model' (Pohl J., A.Chapman, L.Chirica, R.Howell and L.Myers, 1988)
CADRU-03-89 ($12.00)	'ICADS: Working Model Version 1' (Pohl J., L.Myers, A.Chapman and J.Cotton, 1989)
CADRU-04-90 ($12.00)	'Knowledge-Based CAAD and the CLIPS Expert System Shell' (Pohl J., L.Myers, A.Chapman, L.Chirica, J.Snyder, H.Assal, J.Taylor, C.Johnson and D.Johnson, 1990)
CADRU-05-91 ($12.00)	'ICADS Working Model Version 2 and Future Directions' (Pohl J., L.Myers, A.Chapman, J.Snyder, H.Chauvet, J.Cotton, C.Johnson and D.Johnson, 1991)
CADRU-06-92 ($12.00)	'A Computer-Based Design Environment: Implemented and Planned Extensions of the ICADS Model' (Pohl J., L.Myers, J.Cotton, A.Chapman, J.Snyder, H.Chauvet, K.Pohl and J.La Porta, 1992)
CADRU-07-92 ($12.00)	'AEDOT Prototype(1.1): 'An Implementation of the ICADS Model' (Pohl J., J. La Porta, K.Pohl and J.Snyder, 1992)
CADRU-08-93 ($12.00)	'Object Representation and the ICADS-Kernel Design' (Myers L., J.Pohl, J.Cotton, J.Snyder, K.Pohl, S.Chien, S.Aly and T.Rodriguez, 1993)
CADRU-09-94 ($15.00)	'Thoughts on the Evolution of Computer-Assisted Design' (Pohl J., L.Myers and A.Chapman, 1994)
CADRU-10-95 ($20.00)	'Inter-Process Communication in Support of the ICDM Framework' (Pohl K.J., J.Taylor, L.Myers and J.Pohl, 1995)

CAD Research Center, Architecture Dept., Cal Poly, San Luis Obispo, CA 93407, USA; [FAX: (1) 805-756-5986]