1998

Dynamic Process Management in CLAM

Juan Gomez

Vernon J. Rego
Purdue University, rego@cs.purdue.edu

Report Number:
98-012

DYNAMIC PROCESS MANAGEMENT IN CLAM

Juan Gomez
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR #98-012
April 1998

Dynamic Process Management in CLAM

Juan Gomez and Vernon Rego

April 30, 1998

Abstract

The dynamic management of processes is a fundamental task in distributed applications. In its general form it must encompass specialized models of parallel programming such as Task Farming models or Parallel Client/Server models, and must also provide for important features such as fault tolerance and load balancing. But despite the significance of dynamic process management, there is no portable and efficient framework that defines and supports a general functionality, particularly for multithreaded distributed computing. In response to this need, we present a portable and efficient process management module in the context of the CLAM (connectionless, lightweight, and multiway) threads-based system for distributed operations on heterogeneous networks. We present a process management module that supports a wide range of services, including process naming, dynamic process creation, pid-based message routing, process sign-off, system shutdown, I/O redirection, and signal delivery. We describe how CLAM's active messaging interface and its threads are used to build dynamic process management features into the library. For each service, we provide a description and an implementation. Further, we define and implement a formal protocol to support these services. We also present experimental data that compares the overall performance of CLAM's process management services to equivalent services in PVM, LAM-MPI, and P4, using both uni- and multi-processor hosts.

1 Introduction and Motivation

With the emergence of distributed computing via hundreds of communicating processes we have come to witness the importance of scalable and efficient communication [1, 2, 3, 4]. In this paper, we restrict our attention to two key problems, namely, the problems of distributed process management and distributed process scheduling [3, 5]. Many approaches to process management and scheduling have been proposed, and standards continue to emerge [2]. The problem of distributed process management concerns the provision of services like, for example, process creation and initialization, I/O redirection, signal delivery, process naming, system-wide orderly termination, process termination, message routing based on process naming information, and the provision of status and configuration information. The problem of distributed process scheduling concerns the allocation of CPU resources to a particular set of distributed processes. It may involve techniques like Gang Scheduling [6], where an application's distributed tasks are scheduled simultaneously to improve communication performance.

Distributed Multiprocessors (DMPPs) (e.g., Intel Paragon, IBM SP2) provide batch and interactive schedulers, message-passing libraries, and process management tools as an integral part of their operating environment. These services, however, are not readily available for heterogeneous processor clusters (e.g., workstation clusters) and this may be due, in part, to a lack of standards. While various software packages provide mechanisms for process scheduling and management on heterogeneous workstation clusters, these mechanisms are invariably built into the packages, and lack the modularity that encourages flexibility, enhancement, or reuse. Further, because these mechanisms are built into single-threaded libraries, they do not integrate well or cannot be expected to perform satisfactorily with protocols built with threads [4].

Primitive	Description
pvm_mytid()/*pvm_exit*()	Adds/Extracts processes
pvm_spawn()/*pvm_kill*()	Creates/Destroys processes
pvm_parent()/*pvm_pstat*() *pvm_config*()/*pvm_mstat*() *pvm_tasks*()	Configuration and status information
pvm_catchout()/*pvm_getopt*() *pvm_setopt*()	I/O redirection
pvm_sendsig()/*pvm_notify*()	Process signaling
pvm_halt()	Shutdown
pvm_addhost()/*pvm_deletehost*() *pvm_reg_hoster*()/*pvm_reg_rm*() *pvm_reg_rm*(), *pvm_reg_tasker*()	Process scheduling and resource management

Table 1: PVM Process Management and Scheduling Primitives

In some message-passing systems (e.g., PVM [7]), process management and scheduling is done with the help of daemons located on each host in a virtual machine. Table 1 lists primitives that the PVM system uses for process management and scheduling via daemons. The P4 [8] system also provides primitives for process creation, naming, and configuration information, as shown in Table 2; these represent a subset of those provided by the PVM system. The LAM MPI [9] system also uses daemons to provide some of this functionality for process management/scheduling. The MPI Forum, however, recommends that all process management/scheduling functionality be left out of the MPI interface; instead, this functionality should be provided by additional software modules.

There are systems that interface with message-passing libraries for process management and scheduling. DQS [10] provides batch and interactive scheduling for P4 and PVM. Condor [11], a batch scheduler designed specifically for PVM, dispatches and relocates jobs on workstation clusters. The architecture described in [12] clearly defines how message-passing, process scheduling, and process management tasks interact. This work, in particular, emphasizes the need for a clear interface definition between these tasks, and the need for their support in distinct modules for reusability. But this work, however, is based on the single-threaded model of the PVM library, and restricts flexibility and the potential for concurrency. Besides affording a poor and inefficient interface, there is a high cost associated with running process management tasks in an address space that is distinct from an application's space. In effect, the single-threaded model necessitates the use of multiple heavy-weight OS processes (HWPs); their executions overlap with long latency transactions such as those related to the creation of new processes. Several heavy-weight context switches may ensue in a communication between a process responsible for management and a process hosting an application. Many applications, including Task Farms, Parallel Client-Server applications, and, in general, ones in which working process set size and configuration changes dynamically, can greatly benefit from a low-cost interface to a distributed process management module [2].

In this paper we propose and implement an efficient prototype for distributed process management. This system is an intrinsic part of the CLAM (connectionless, lightweight, and multiway) communications architecture [4, 13]. Here, the idea is to hide communication latencies and reduce code complexity with the help of user-space threads. By integrating three components—the communications library, the process management module, and the application—and placing them within a single address space, we show how a clean and low-cost interface can be built to provide for efficient interaction between these modules. By replacing HWPs with user-level threads, long-latency transactions that are generally associated with some process management operations are easily overlapped with computations. This

Primitives	Description
p4_initenv()/p4_create_procgroup() *p4_alloc_procgroup()/p4_startup()*	Process creation
p4_get_my_id()	Process naming
p4_wait_for_end()	Process termination
p4_num_total_ids()/p4_num_total_slaves()	Configuration information

Table 2: P4 Process Management Primitives

is done at the simple expense of a user-level thread's context switch. Such features make CLAM an attractive platform for distributed applications involving fault tolerance, load balancing, collaborative interaction, and, in general, in situations where there is a frequent need for invocation of process management tasks (e.g., the creation or destruction of processes).

For completeness, we formally define the application protocol that supports all the process management features provided by CLAM. This protocol may be used and enhanced by other systems. CLAM's process control module offers the following functionality: process naming, pid-based routing, I/O redirection, signal delivery, process and system-wide termination, and status query. At this time, CLAM does not provide process scheduling services. In the experiments reported here, a static process scheduler programmed in Perl [14] was used. The scheduler initiates processes on hosts that are lightly loaded at startup time.

The remainder of the paper is organized as follows. In Section 2 we describe the CLAM environment, its process hierarchy, and its hierarchical routing. The protocol that supports CLAM's dynamic Process Management Interface is described in Section 3, along with the details on signals, process creation, and process termination. In Section 4 we describe the initialization procedure based on a host-file. We present a set of experimental results in Section 5 and conclude briefly in Section 6.

2 The CLAM Environment and Process Hierarchy

The Connectionless Lightweight and Multiway (CLAM) [15] communications environment is a software architecture designed to support interactive, multiprotocol, and multithreaded distributed computing on heterogeneous workstation networks. CLAM is layered on top of the ARIADNE [16] user-space threads library and supports multithreaded (distributed) applications. CLAM uses threads to improve performance through an overlap of communication with computation and an efficient integration of functionality [3]. The use of threads enables a reduction in the complexity of code required for integrating an application with multiple protocols, inside a single address space. Through specialized adaptive scheduling algorithms [13, 17], CLAM's threads distribute CPU time fairly across an application and its (multiple) protocol modules. This strategy makes it ideal for distributed multimedia applications where progress must be guaranteed for each protocol session.

The basic structure of the system is shown in Figure 1. In its simplest form, it includes three protocol modules and a process management module. The basic protocol modules include: a transaction-oriented and reliable point-to-point protocol (TRAP) [4], a transaction-oriented reliable multipoint protocol (TRAM) [18], and an unreliable point-to-point and multipoint module. Each basic protocol module provides support for both passive messages and active messages. The passive message interface of these protocols resembles a message-passing interface. Active messages are supported in two forms: proper Active Messages (AMs), as described in [19], and Remote Thread Activations (RTAs) [4]. While AMs are more efficient than RTAs, they have limited flexibility because they cannot invoke thread primitives that may potentially block.

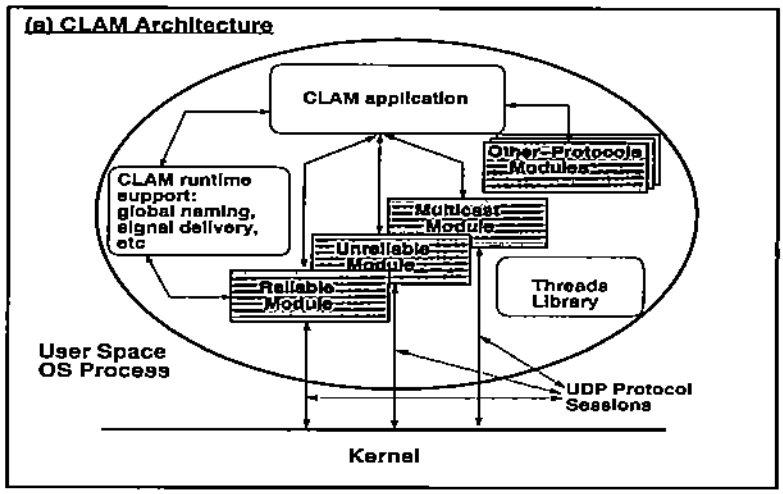


Figure 1: CLAM's Structure

Although CLAM protocols can be layered on top of any best-effort communication subsystem, we have employed UDP in our prototype implementation. One of the main advantages of using UDP instead of TCP/IP consists in a significant reduction in the number of protocol sessions needed to maintain a distributed computation. While TCP/IP requires $N(N-1)/2$ protocol sessions to keep a fully connected distributed computation involving N processes, UDP (and a reliable protocol implemented on top of it) only requires N protocol sessions. Besides reducing the usage of kernel resources, this approach also diminishes the overhead of user-kernel interactions involved in communicating one process with multiple destinations. This performance improvement is related to the movement of the multiplexing and demultiplexing functionality to user space and not to the use of specific protocols. In fact, about the same improvement could be achieved if the socket interface to the in-kernel TCP/IP protocol would allow connecting a single socket to multiple remote end-points. Other factors that contribute to an efficient point-to-point communication using UDP include the elimination of the three-way hand-shake for connection establishment, and the introduction of a tight integration between the application and the protocols. UDP also provides portability, makes our system easy to deploy, and enhances flexibility by freeing us from the stream-oriented reliable model of TCP/IP.

2.1 The Environment

CLAM's processes are structured hierarchically and classified according to their function. There are five types of processes. Some are used, in part, to support the runtime Process Management Interface. They help during the startup procedure, in routing messages to destinations, and also act as multicast routers when there are no other means to forward multicast packets. There are three types of processes in this category: HyperText Transfer Protocol (HTTP) servers, Fork Servers (FS), and Domain Servers (DS). The other two types of processes are related to user computation and may or may not include the CLAM library code. OS-level processes that load the CLAM library and run application code are called Computing Processes (CP). Other user level processes in a CLAM session—which do not contain CLAM code—are simply called User Processes (UP). Although CLAM processes (i.e., DS, CP) can communicate with UPs, such communication is not supported by the library and must be handled by the application.

HTTP servers are used exclusively during the startup procedure in order to efficiently create processes in hosts where servers run. The Fork Servers (FS) are used when no HTTP server is available on

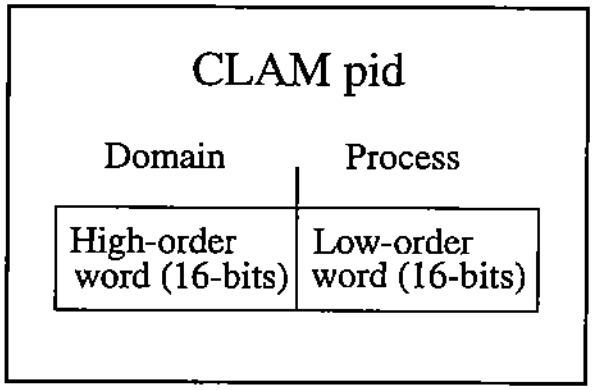


Figure 2: Structure of CLAM's Process Identifiers

a given host. The FS processes reduce overheads related to *rsh* or *rexec* invocations; these invocations are, in general, more time-consuming than corresponding CLAM protocol invocations handled by an FS process. FS processes are state-less, with functionality limited to receiving and executing spawn requests on behalf of remote processes.

Domain Servers (DS) are CLAM processes with key functionality in the CLAM environment. These are assigned a CLAM pid, can communicate using CLAM protocols, and can help in routing messages. If a CLAM process cannot map a given pid into a low-level protocol address using local information, DS processes help forward the message to its final destination. DS processes also aid in decentralizing process management functions and data, thus increasing scalability and fault tolerance. In the absence of regular multicast routers, DS processes assume routing responsibility, and optimize collective communication across a WAN or a set of interconnected LANs. While DS processes are primarily meant to manage CPs in their own IP or physical networks, a DS process can efficiently manage CPs that are distributed across several locally interconnected networks. Several DS processes may co-exist in one IP or physical network. This is beneficial when the number of CPs is large, since a single DS process may become a bottleneck. With threads support, management tasks done by DS processes are transparent to applications that share the same address space. Using DS processes to run part of a distributed application may, however, affect the performance of process management and reliable multicast across different domains.

There are two kinds of DS processes: Root Domain Servers (RDSs) and Regular Domain Servers (DSs). The RDS is a DS that serves as a communication hub for all regular DS processes; it also assigns pids to DS processes. There is only one RDS in a CLAM session, and this process is always assigned the well-known pid of zero (0). CLAM processes are organized hierarchically under the RDS. The RDS routes messages that cross domains whose members have not communicated previously.[1] If a process does not know how to communicate directly with another, it uses its own DS—and possibly the RDS—to locate its target based on a pid. For each such pair of communicating processes, only one RDS intervention suffices. Subsequent messages use locally installed pid-to-protocol-address mapping entries after the first message and RDS intervention.

In CLAM, processes refer to one another using integer-valued process identifiers (pids). CLAM pids have two parts: a domain identifier and a process identifier (see Figure 2). This structure enables pid-based routing across different CLAM domains. If the sender is unable to map a message's destination pid to a low-level protocol address, the message is given to a DS, which forwards it based on a destination pid. Each DS manages its own pid space independently, thus facilitating dynamic process management.

[1] In CLAM, a domain is a group of CPs and their corresponding DS.

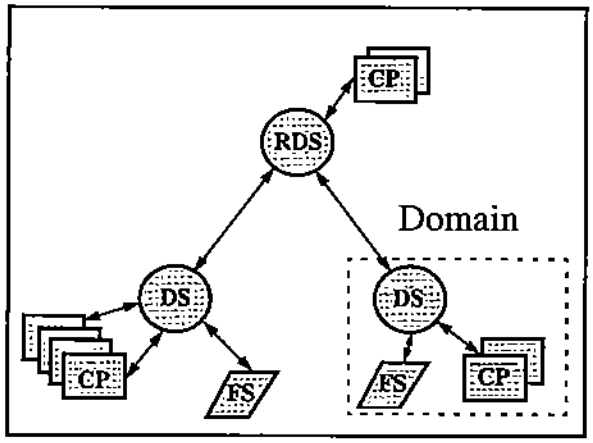


Figure 3: CLAM Process Hierarchy

2.2 Process Hierarchy and Routing Tables

In principle, no routing is necessary in CLAM: two processes communicate directly using only one UDP session (socket) per process. To initiate communication, however, processes need to know IP addresses and UDP port numbers.[2] The approach used in most distributed computing systems (e.g., PVM, LAM MPI) is to build a table that maps pids to low-level protocol addresses and distribute the table, during setup, to all participating processes. The main disadvantage is that scalability is affected. Startup times may become prohibitively large because of overheads incurred in distributing information. Dynamic updates of these tables, when possible, must be governed by a central process that becomes a potential bottleneck. Because modifying and redistributing centralized mapping tables to all participants is costly, process management primitives that require modified process tables become very expensive. This severely hampers the scalability of applications that may require dynamic process allocations (e.g., the addition/deletion of participants in a collaborative interaction).

CLAM's hierarchical organization circumvents most of the problems encountered by centralized and static approaches. Processes are organized in domains according to their physical location and patterns of communication. Each domain employs a DS to represent its naming and management authority. Processes start with minimal information in their process tables and update these tables on-demand, as the communication unfolds. Instead of keeping table entries identifying all other processes in a session, a process keeps only those entries that correspond to processes with which it must directly communicate. This reduces process table size and, in the worst case, yields a table that is as large as tables in the usual approach. Observe that the worst case occurs when the communication graph is complete.

Even for applications with very demanding communication patterns (e.g., almost complete graphs) direct links may not be required between all nodes simultaneously; aging and/or recycling algorithms can be used to deallocate old/unused process table entries. After the first (indirect) communication between two processes, subsequent communication is direct, and there is no mediation or routing because of the automatic table build-up feature described below. Dynamic changes in one domain are transparent to other domains because management functionality is distributed. In addition, CLAM's hierarchy also reduces startup times: minimal information needs to be given to each newly created process.

CLAM employs a tree-shaped hierarchy (see Figure 3). The RDS, located at the root, routes information across different domains when processes from these domains initially communicate. The RDS acts as a DS when it interacts with a CP or an FS that it directly manages. It may also manage

[2]This may be any low-level addressing information pertinent to the underlying communication layer.

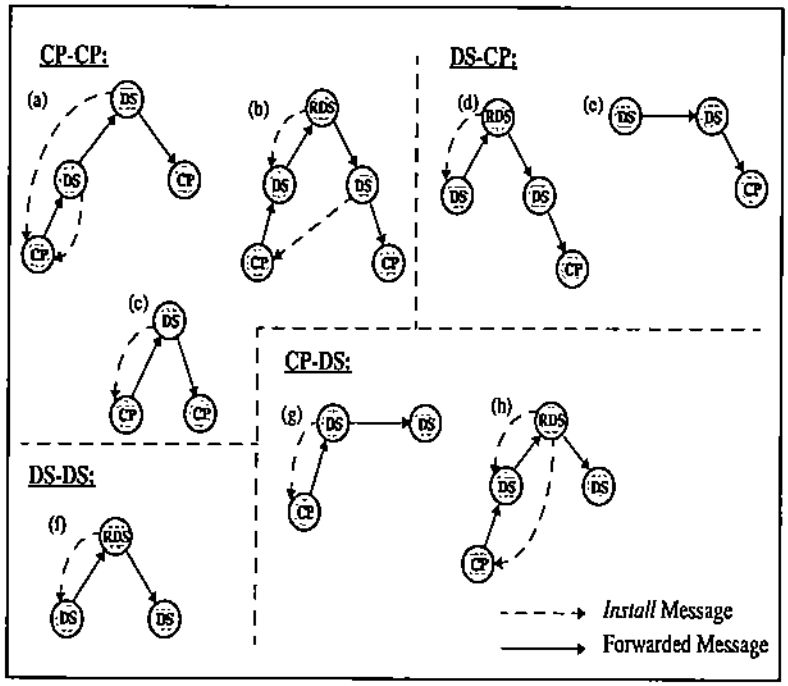


Figure 4: CLAM Routing and Process Tables Configuration

several DS processes. In like manner, each DS may manage a set of CP and FS processes. The RDS manages the assignment of Domain Identifiers, and DS processes assign pids to subordinate CP processes. Each FS is assigned a pid that is equal to its IP address. With the exception of the RDS, each CLAM process has a *master*, which represents its parent in the tree-hierarchy. CLAM processes keep a mapping table for each process type (i.e., DS, CP, FS). These tables help processes map pids to low-level protocol addresses, enabling point-to-point communication; initially empty, the tables are updated as processes begin to communicate during setup.

The RDS is the first process created. Its low-level protocol address is the only information required in the process table of a DS that wants to enroll in a CLAM session. The RDS begins with a single entry in its process tables: its own pid. Both CP and FS processes are subordinate to their parent DS and, to join in a CLAM session, they require only the address of their masters (i.e., their DS processes). CP and FS processes may run only after the DS that manages their domain is initiated, since their communication depends on this DS. A DS updates its tables when a new CP enrolls; it may also create new FS processes and add corresponding entries to its tables during system startup, or later during the session. This dynamic updating feature requires that each CP and FS must sign-off with its master DS when it leaves a CLAM session. A DS that leaves a CLAM session must also sign-off with the RDS process and all the processes it manages before it exits.

2.3 Hierarchical Routing

When a CLAM process sends a message to a process whose address is not in its local tables, the message is forwarded to the destination (see Figure 4). But any uncontrolled forwarding of messages through these paths, however, may swamp a DS with excessive routing load. This will occur, for example, when a source generates a burst of large packets, or when multiple sources in one domain generate many

packets simultaneously. To avoid such chaotic scenarios, only the first message sent from a process to one it is not directly connected to uses the routing shown in Figure 4. This initial routing results in the installation of an entry in the source's local table; the entry enables subsequent communications between the process pair to be direct. But even the initial message's routing via Domain Servers may be problematic. Because different messages may follow different paths, and because there is no guarantee that the initial message has reached its destination when the direct path is enabled, the message routed via a DS may, in principle, arrive out-of-order. Also, there is no limit on the amount of data the first message may contain, and multiple sources that send large messages through the same DS may overwhelm it with routing work.

To avoid problems, only *Null* Active Messages (AM) are forwarded. Such a message only triggers the installation of a process table entry at a source, enabling the latter to communicate directly with the given destination. CLAM's routing infrastructure is designed to handle messages with arbitrarily large sizes; this feature is crucial in implementing reliable multicast across multiple domains. Null AMs are very small and are quickly forwarded by DS processes. Even if a DS receives many such AMs simultaneously, flooding is unlikely unless the size (in members) of its domain is very large. In this case, the addition of new domains will resolve the problem.

If a sender has a table entry for a direct path to a destination, its messages to this destination are sent immediately. Otherwise, the sending process issues a Null Active Message and blocks the sending thread until a direct path is created. Any thread that attempts to send while the direct path is being installed will also block until the path is established. This path is created as a side effect when the Null AM is forwarded to its destination. Forwarded messages are tagged as self-routing and sent as special Optimistic Active Messages (OAMs) [20] so that they obtain the special and efficient processing they require. Using OAMs for routing messages also helps prevent deadlocks. If a process's receive thread is allowed to block on flow- or congestion-control, while trying to forward a packet, then flow- and congestion-control windows cannot be opened because acknowledgments and new messages cannot be received; the result is a deadlock. Using an OAM for forwarding guarantees execution of the forwarding procedure on the stack of the receive thread, when flow- or congestion-control allows forwarding without blocking. If blocking is likely during the routing procedure, a new thread is created to forward the message while the receive thread continues to run without blocking. Through this, we avoid thread context switching overheads for each forwarded message whenever possible. Self-routed messages forward themselves through the CLAM hierarchy based on the destination pid held by the message and local information contained in the process tables of a forwarding process.

Message forwarding (e.g., like the one undergone by Null AMs) through paths such as those shown in Figure 4 proceeds as follows. If a sending process has a direct path to the DS managing a destination CP's domain, the message is forwarded to the DS, and the DS forwards it to the CP. If the sender is a CP that does not have a path to the destination's DS, it forwards the message to its own master (i.e., its own DS). If neither of these two cases is true, and if the sender is a DS but not the RDS, the message is simply forwarded to the RDS. Finally, if the process initiating or routing the message is the RDS, and a local path to the destination's DS does not exist, then an error has occurred.

Clearly, if the routing described above is used for every message, communication would be inefficient. In practice, this can be observed in daemon-oriented systems like PVM. Applications based on daemon-based routing achieve about half the communication throughput when compared to direct TCP/IP connections [7]. Fortunately, CLAM requires only one message to be forwarded before pid tables are dynamically configured and all future routing between the process pair in question is eliminated. This dynamic self-configuration feature enables direct point-to-point communication after a Null message or a regular message is sent between two CLAM processes.

Whenever a CLAM process forwards a message that originated elsewhere, and for which a destination pid entry exists in local tables, the entry is installed at the message's source. The only restriction is when a source is a DS and the destination is a CP; such a restriction is imposed in order to prevent the

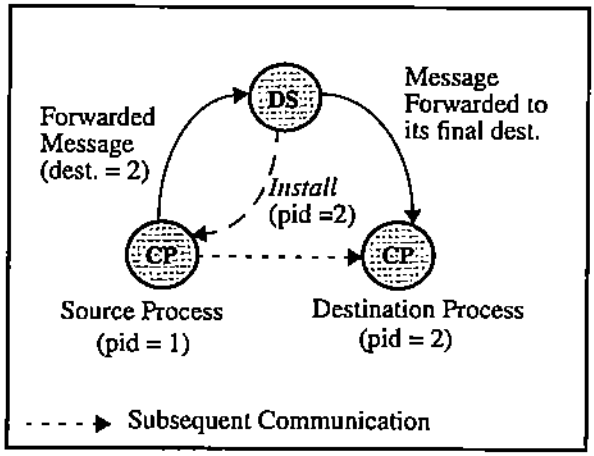


Figure 5: CLAM pid-based Routing: an Example

installation of CP entries in the tables of foreign DS processes.[3] Forcing a DS to collect information on CP processes in different domains inhibits scalability because mapping tables at a DS will grow large. Also, this information becomes stale as domains change configuration and foreign DSs do not get notified.

If a message being routed by a process originated elsewhere, and a DS—either the final destination or the destination process's DS—is located at the next calculated hop, then an entry identifying this DS is installed at the previous hop.[4] This rule enables the RDS to install DS entries in the tables of other DS processes when the RDS routes messages across domains. A DS determines the low-level protocol addresses of all the other DS processes through entries installed by the RDS. This rule also permits a DS to install entries identifying DS processes in the tables of CP processes in its own domain. By allowing a CP to install process table entries that directly lead to other domains, efficient communication is enabled between a given CP and CPs in other domains.

Using the rules described above, each sending process learns low-level protocol addresses of destinations quickly and with minimum overhead. In like manner, Domain Servers that govern communicating domains quickly learn low-level protocol addresses to reduce forwarding effort. As shown in Figure 4, this scheme involves a single *Install* message that is generated by each forwarding operation for most communication paths that require routing. Configurations (b), (d), and (e) in Figure 4, however, require fewer Install messages.

The installation of entries in remote process tables is done by an Install Remote Thread Activation (RTA) message generated by a routing process. This Install RTA carries the pid and low-level protocol address of the CLAM process to which the message was forwarded. When the Install RTA reaches its destination, which is usually the source of the message, it installs an appropriate entry in the corresponding process table. The next time this source sends to the same destination, direct point-to-point communication is used. Thus, after the first message has been sent and some time has elapsed, the sender is able to communicate directly with the destination. The elapsed time is the time taken by the Install RTA message to reach the sender and effectively install information in the sender's tables.

To demonstrate how entries are installed in process tables, consider the example shown in Figure 5. Assume that two CP processes within a domain want to communicate. Initially, these processes do not know each other's low-level protocol address. Both processes, however, know the low-level address of

[3]A foreign DS is a DS that handles a domain that is distinct from the domain of the CP under consideration.
[4]The previous hop is the process that forwarded the message to the one doing the current routing.

their common DS. When one CP sends a message to the other, local routing forwards a Null message—indicating the final destination—to the DS. At the DS, the message routes itself to the target CP. During the routing, an Install RTA message is sent to the source CP, giving it the (`pid`, `IP address`, `UDP port`) triplet that was used by the DS to forward the message to the destination. When the Install RTA message arrives at the source, it installs an entry in the source's local tables. Thereafter, communication between the source and the destination is direct. While the source may receive multiple Install RTA messages containing information corresponding to a single pid, only the first is significant. When process tables are to be updated, old entries must be deleted before new information is installed. Table entries must be deleted when a process exits a CLAM session. This is accomplished with the help of a *Remove* RTA message.

For applications with well-defined communication patterns, explicit construction of routing tables can be done with the `c_instpa()` primitive. This primitive installs pid-to-low-level protocol address mapping entries at remote processes. It may be used by any process to install pid mapping entries that it knows on remote process tables. Domain Servers, for example, know each of the CP processes in their domains and can install corresponding entries at any remote process. This enables these remote processes to communicate directly with the corresponding DS's CP processes. For applications with all-to-all communication patterns and a static process configuration, this is an efficient way to build process tables.

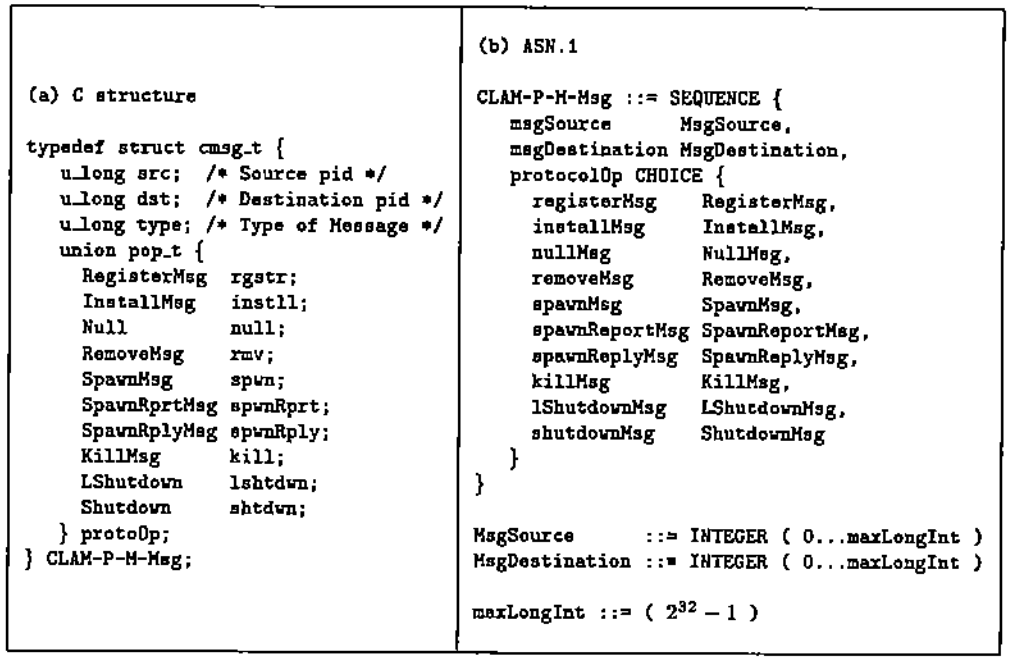
3 The Dynamic Process Management Protocol

A session of distributed processes is managed with the help of a protocol. We present a formal description of CLAM's protocol for dynamic process management, along with an efficient and portable implementation. The protocol is layered on top of the reliable Active Message and Remote Thread Activation transport services provided by the Transaction-oriented and Reliable Point-to-point protocol (TRAP) [4] module in CLAM. The protocol is simple and consists of ten message types: the Register message, the Install message, the Null message, the Remove message, the Spawn message, the Spawn-Report message, the Spawn-Reply message, the Kill message, the Lightweight-Shutdown message, and the Shutdown message. The Register, Install, and Null messages let one process know another's protocol address, to enable direct communication. The Remove message extracts addressing information from process tables. The set of Spawn-related messages is used to create new processes and report the status of spawning operations. The Kill message delivers signals to processes. The Lightweight-Shutdown and Shutdown messages are used for process termination.

3.1 Protocol Description

Although the description of a protocol in terms of a sample implementation may serve as a protocol specification, such a description may be ambiguous and hinder enhancements to the protocol. For example, this may occur if new architectures use different word sizes or different basic data type representations. Complete generality, however, comes at the price of performance because universal intermediate encoding/decoding schemes must be used to compensate for differences in data representations. Because of this we present both a formal protocol description as well as an efficient implementation.

To describe the structure of each message unambiguously we use a language that is capable of representing a wide range of abstract data types, including simple and structured data types. The OSI's [21] Abstract Syntax Notation One (ASN.1) [22, 23] has become a standard for specifying high-level application protocols. ASN.1 is a notation or language used to describe abstract data types and values. It can be used to represent data types that consist of a finite or an infinite set of values. ASN.1 can be used to represent simple data types (which are atomic), structured types (which have components), tagged types (which are derived from other types), and *other* types, including the `CHOICE`



(a) C structure

```
typedef struct cmsg_t {
   u_long src;  /* Source pid */
   u_long dst;  /* Destination pid */
   u_long type; /* Type of Message */
   union pop_t {
     RegisterMsg  rgstr;
     InstallMsg   instll;
     Null         null;
     RemoveMsg    rmv;
     SpawnMsg     spwn;
     SpawnRprtMsg spwnRprt;
     SpawnRplyMsg spwnRply;
     KillMsg      kill;
     LShutdown    lshtdwn;
     Shutdown     shtdwn;
   } protoOp;
} CLAM-P-M-Msg;
```

(b) ASN.1

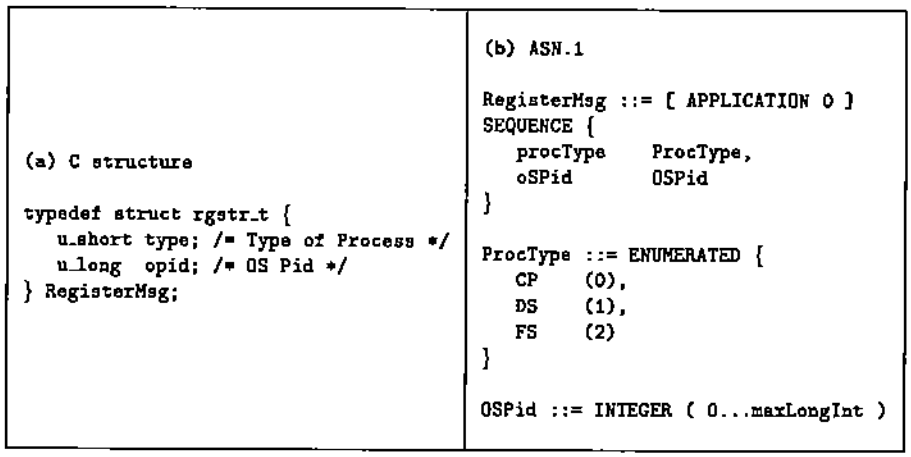
```
CLAM-P-M-Msg ::= SEQUENCE {
   msgSource      MsgSource,
   msgDestination MsgDestination,
   protocolOp CHOICE {
     registerMsg    RegisterMsg,
     installMsg     InstallMsg,
     nullMsg        NullMsg,
     removeMsg      RemoveMsg,
     spawnMsg       SpawnMsg,
     spawnReportMsg SpawnReportMsg,
     spawnReplyMsg  SpawnReplyMsg,
     killMsg        KillMsg,
     lShutdownMsg   LShutdownMsg,
     shutdownMsg    ShutdownMsg
   }
}

MsgSource      ::= INTEGER ( 0...maxLongInt )
MsgDestination ::= INTEGER ( 0...maxLongInt )

maxLongInt ::= ( 2^32 - 1 )
```

Figure 6: CLAM Process Management Message

and ANY types. Types and values are assigned through the equal (::=) operator. Bold braces ({}) are used to group related terms; brackets ([]) are used to indicate optional terms. A vertical bar (|) is used to delimit alternatives in a group of related terms. The subset of simple types defined by ASN.1 that are used in this document include IA5String, which represents an ASCII string, INTEGER, which represents a regular integer, and ENUMERATED, which is a type with a finite and well-defined set of possible values. The two structured types defined by ASN.1 that are relevant to this description include SEQUENCE and SEQUENCE OF. The SEQUENCE type defines an ordered collection of one or more types. The SEQUENCE OF defines an ordered collection of zero or more occurrences of a given type. The CHOICE type denotes the union of one or more alternatives. Tagged types are those that are assigned a specific number. Tags may have an abstract meaning in the context of an application, an enterprise, a given context, or they may have a universal abstract meaning. Only application-specific tags are used in this description.

The OSI framework also recommends the use of the Basic Encoding Rules (BER) [24] to encode each ASN.1 value as an octet string. BER, however, may offer multiple representations for a given value; hence, an encoding ambiguity results. A subset of the BER rules, namely, the Distinguished Encoding Rules (DER), offers a unique encoding for each ASN.1 value and has become popular in protocol specifications. Although this encoding/decoding scheme allows protocol specifications that are completely architecture independent, we have not used these rules in our current protocol implementation to maximize efficiency and simplicity. Routines for encoding/decoding DER data only add latency and other overheads to the protocol. The protocol we describe uses only unsigned long integers, unsigned short integers, and ASCII strings. Conversion routines or macros, available in the standard networking library, may be used to convert these types from and to network-byte order. As a result, more complex data representations are unnecessary. On the other hand, the ASN.1 notation enhances our description of the protocol by unambiguously describing the structure of messages. Furthermore, a formal description of the protocol represents a solid base for future enhancements to it.

A CLAM Process Management message has the general structure depicted in Figure 6. There is a



(a) C structure

```
typedef struct rgstr_t {
   u_short type; /* Type of Process */
   u_long  opid; /* OS Pid */
} RegisterMsg;
```

(b) ASN.1

```
RegisterMsg ::= [ APPLICATION 0 ]
SEQUENCE {
   procType    ProcType,
   oSPid       OSPid
}

ProcType ::= ENUMERATED {
   CP    (0),
   DS    (1),
   FS    (2)
}

OSPid ::= INTEGER ( 0...maxLongInt )
```

Figure 7: *Register* RTA Message

common header that contains the CLAM pid of the source and destination of the message. The type field in the "C" description (Figure 6(a)), which defines the content of the union field, is represented in the ASN.1 description as an application tag. The structure of all the messages and the actions taken upon their arrival at their destinations are described next.

3.2 Process Table Initialization

The *Register* and *Install* messages are used to initialize and build process tables in each CLAM process. These tables contain information that maps CLAM pids into low-level protocol addresses. A process initializes its routing tables by registering with its master process through the Register Remote Thread Activation (RTA) message. The format of the Register RTA is shown in Figure 7. In Figure 7(a) is shown a "C" language description, and in Figure 7(b) is shown an ASN.1 representation. The type field in the "C" structure specifies the type of process (i.e., DS, CP, FS) that issued the Register RTA. The opid field is its OS pid. When the Register RTA arrives at the master process, three actions are taken. First, a CLAM pid is reserved for the registering process. Second, an entry is installed locally, in the process table of the master; the pid identifying this entry corresponds to the slave that sent the Register RTA. Third, the master sends an Install RTA back to the slave that sent it the Register RTA in the first place.[5] When the Install RTA arrives at the slave, two entries are installed in its tables: an entry that corresponds to the master, and an entry for itself—the latter containing the newly assigned pid. Because several DS processes may register simultaneously with the RDS, and because several CP processes may register simultaneously with a given DS, the global variables used for pid and domain id assignment must be protected by locks.

A process obtains its master's low-level address through its command-line arguments. Except for the RDS, all CLAM processes must be given this information via the `-R` command-line switch. A process that is not manually started installs an additional entry in its tables; this entry corresponds to its *parent*. In CLAM, the parent of a process is the process that generates the *Spawn* request creating the process. Observe that, in some cases, the parent may be the master itself. A parent's low-level addressing information is passed to the child as an optional command-line argument (`-S`) during initialization. If the process has a parent, then the parent also installs an entry corresponding to its child.

[5]Note that a CLAM pid is reserved by the master process only if the process that is registering is not a Fork Server (FS).

(a) C structure

```
typedef struct inste_t {
   u_long  pid;  /* CLAM Pid */
   u_long  ospid;/* OS Pid */
   u_short type; /* Type of Process */
   u_long  ip;   /* IP Address */
   u_short port; /* UDP port */
} InstallEntry;

InstallEntry InstallMsg[ ];
```

(b) ASN.1

```
InstallMsg ::= [ APPLICATION 1 ]
SEQUENCE OF SEQUENCE {
   pid      Pid,
   oSPid    OSPid,
   procType ProcType,
   iPAddr   IP,
   port     Port
}

IP   ::= INTEGER ( 0...maxLongInt )
Pid  ::= INTEGER ( 0...maxLongInt )
Port ::= INTEGER ( 0...maxShortInt )

maxShortInt ::= ( 2^16 - 1 )
```

Figure 8: *Install* RTA Message

The format of the data encapsulated in an Install RTA message is shown in Figure 8. Each Install RTA message may contain multiple entries that must be installed at a destination process. The information carried by each entry in this RTA message includes the CLAM pid (`pid`), the OS pid (`ospid`), the type of process (`type`), and the low-level protocol addressing information containing the IP address (`ip`) and the UDP port number (`port`). Once an Install RTA message reaches its destination, it proceeds to install all entries contained in the message that do not already exist in the destination's local tables. In Figure 9 are shown the RTA messages involved in a startup operation.

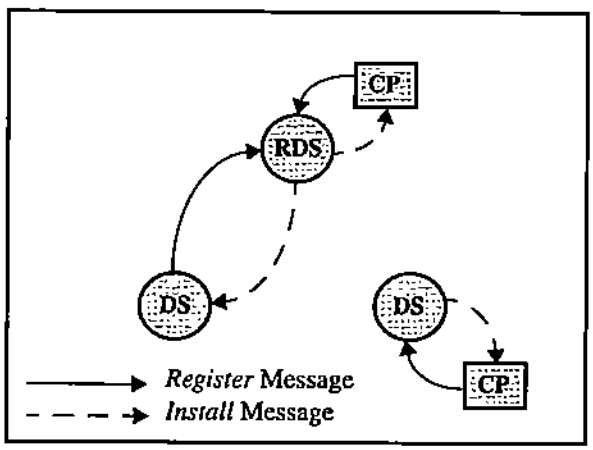


Figure 9: CLAM Initialization

During initialization, each CLAM process but the RDS blocks until the first Install RTA message is received. This ensures that the CLAM communication primitives are used only after process tables obtain the basic information.

3.3 Process Creation

CLAM's Process Management Interface (PMI) provides a complete set of calls to create processes dynamically anywhere in the Internet, as long as the user has the required permissions. The interface can be used by a job scheduler, a user console, a startup routine, or by an application to create new CLAM processes. The PMI takes advantage of multithreading and non-blocking system calls to concurrently process high-latency invocations of *rsh*, *rexec*, and other remote access mechanisms. These mechanisms exhibit high-latencies because of their implicit use of the TCP/IP protocol, with its three-way handshake connection establishment procedure. Multithreading helps overlap these latencies with other useful work, thus providing for improved performance and scalability.

3.3.1 Basic Startup Mechanisms

Most message-passing systems use *rsh* or *rexec* to create processes on remote machines [7, 8, 9]. Some systems also provide *startup daemons* to reduce the high latency and overheads due to *rsh* and *rexec*. These startup daemons are processes that exist on every host in the distributed environment; their main function is to receive and process startup requests locally. The daemons respond to startup requests by forking new OS-level processes and loading specified code. In general, these daemons are initially created with *rsh*, and they may be reused over several runs of the same or a different application. While their use reduces startup time, they are tied to a particular message-passing system because of the distinct format of their startup requests. Thus, even though startup daemons for different message-passing systems have the same basic functionality, one system's daemons cannot be used by another system.

We propose another alternative to the methods mentioned above: the HyperText Transfer Protocol (HTTP) [25], a de-facto standard for access to Internet resources. This protocol can be used for efficient process creation. By using existing HTTP servers for process creation, we are saved the task of initializing CLAM's Fork Servers (FS). Besides, HTTP daemons can serve other purposes as well. This technology is well-supported and there is much ongoing research geared toward the provision of faster and more efficient HTTP servers based on threads. By exploiting HTTP in CLAM's PMI, the user can also take advantage of existing Web browsers to interact with a CLAM session. In contrast, other message-passing systems usually provide a proprietary graphical user interface. Yet another advantage of the HTTP protocol for process startup is its interoperability with threads. In traditional methods for remote process creation (e.g., *rsh*, *rexec*), a new process must be forked to prevent the parent from blocking while a creation request is handled.[6] With HTTP, non-blocking system calls and new user-level threads can be used to handle new startup requests, thus overlapping one request's latency with another request's computation.

To use a HTTP server for process creation, the software must be installed and configured on every target host. But this is a one-time task, and these daemons can be left in place to serve a group of users, as is the case with PVM daemons. All the CLAM process's code must be marked as executable and accessible to the HTTP daemon. Processes can be started up interactively with the help of a regular Web browser that reads a HyperText Markup Language (HTML) [26] page with the URLs of all target hosts, applications, and parameters that must be passed to remote processes through the Common Gateway Interface (CGI) [27]. CLAM processes can configure themselves in a group by using a unique, global process that listens at a well-known address. The latter process enables all the other processes to get to know about one another. Other methods to create processes (*rexec, rsh*, etc.) may also be made available to the user of a Web browser by building scripts or programs that use CGI to receive their parameters. These scripts can then execute the required commands or system calls from a central host.

[6] Although this may not be necessary in systems that support kernel threads, not all our target operating environments support this feature.

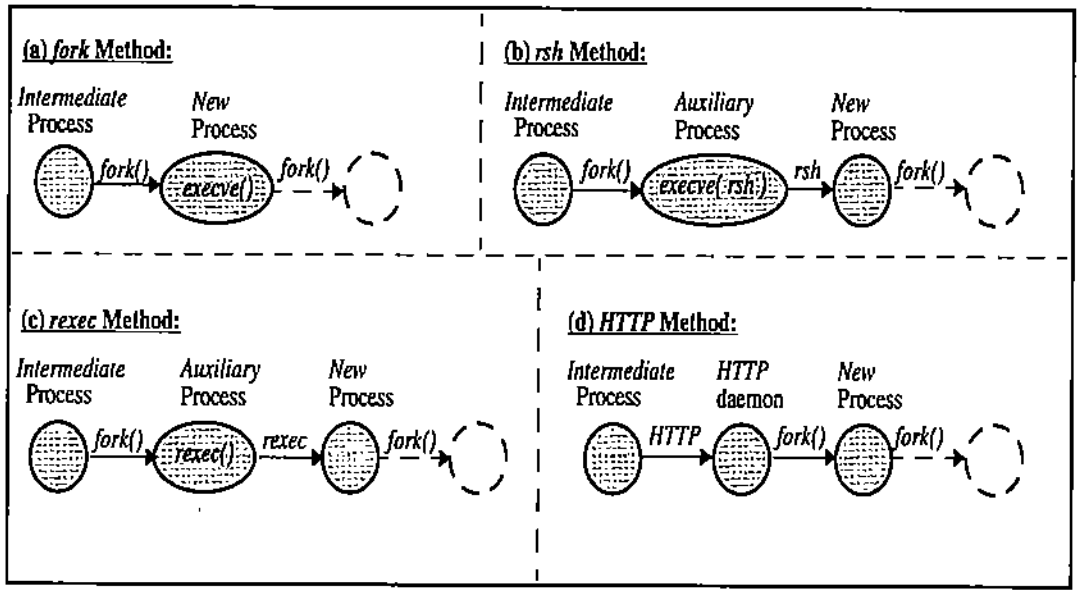


Figure 10: Processes Involved in Each Startup Method

To handle situations in which the user may not be willing or able to use HTTP, CLAM also supports well-known startup methods based on *rsh* and *rexec*. CLAM's startup mechanisms include *rsh*, *rexec*, *fork*, and HTTP. The *rsh* command is used to create processes at hosts in which no CLAM process or HTTP daemon already runs. The *rexec* method is used when *rsh* is not available or cannot be used. The *fork* method is used when a CLAM process already exists on a remote host. When this process receives a startup request, it executes a fork system call and then loads the code of the new process.

There are many ways to use CLAM's PMI to create a group of distributed processes. The user may specify details of the session, e.g., the number of processes, where and when they must run. A job scheduler then locates the specified hosts and determines when the session begins [10, 11]. Alternatively, the user can select the hosts to be used in a specific session and save that information in a file before actually initiating the session. At runtime, this file is passed to a startup program that executes all the necessary steps to create the processes specified in the file [7, 8]. Finally, the user can create the processes interactively from a console. The first and second methods can be implemented in a script language (e.g., *Perl* [14]) or a shell. An interactive startup interface can be implemented with a regular HyperText Markup Language (HTML) browser and Common Gateway Interface (CGI) scripts.

3.3.2 Details of Process Creation in CLAM

A typical startup procedure in CLAM involves three types of processes: a *parent* process, which issues the request, an *intermediate* process, which does the startup, and a *new* process, which is created in response to the request. Some startup methods (e.g., *rsh*) use an *auxiliary* process to prevent an intermediate process from blocking while a new process is being created. Because the *rexec* library call and the *rsh* command generate high-latency blocks on internal sockets, creating a new OS-level process for every *rsh* and *rexec* prevents blocking delays. The *fork* and HTTP startup methods do not need an auxiliary process to hide latencies. Each HTTP-based request is processed by a distinct thread that is created exclusively for this purpose. The sets of processes involved in the different startup methods are shown in Figure 10. In each figure a broken arrow represents a potential fork operation; a CLAM

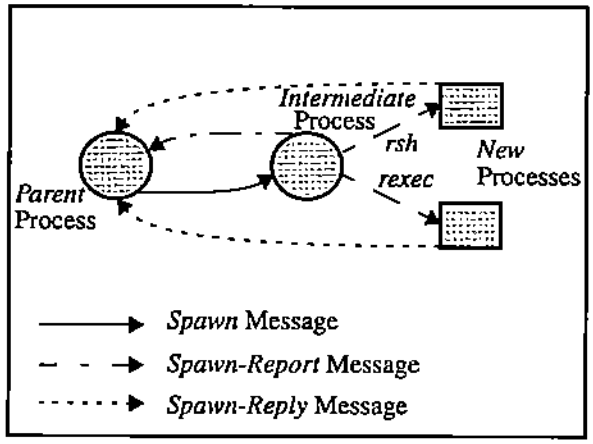


Figure 11: Messages Involved in CLAM's Process Spawning

process may detach itself from its parent by closing its standard file descriptors and creating a new process. The broken circle in Figure 10 represents this new process. For simplicity, we have illustrated cases with only a few processes, even though more OS-level processes may be involved in a startup.

A startup request is generated when a (parent) process invokes the `c_sndspwn()` function. Function parameters specify the number of processes to be created, the creation method, the CLAM pid of the intermediate process that does the job, and other configuration details. The parent thus generates a *Spawn* Remote Thread Activation (RTA) message addressed to the intermediate process.[7] Upon receiving the request, the intermediate process initiates the new process set and generates a status report for the parent (using a Spawn-Report RTA message). Finally, each new process reports to the parent with a *Spawn-Reply* RTA message. In Figure 11 a startup mechanism is shown for a specific request. A parent asks an intermediate process to start two new processes, one with an *rsh* and another with an *rexec*.

The intermediate process collects information on the status of each spawn request. This information is sent to the parent when all requests within a Spawn RTA have been processed. In the case of *fork*, *rsh*, and *rexec*, the intermediate process uses pipes to communicate with newly created processes. It reads from these pipes to obtain status information. To enhance concurrency and hide latency while multiple processes are created, pipes or sockets connecting the intermediate process to auxiliary or child processes are read with non-blocking system calls. The read terminates on a timeout (initiated by a new process's creation) or when the remote end of the pipe is closed. When all reading related to one spawn request has terminated (i.e., all pipes are closed), the intermediate process sends its information to the parent with a *Spawn-Report* RTA message. A newly created CLAM process detaches itself from a parent by redirecting `stdout`, `stderr`, and `stdin`, and forking a new OS-level process. A user may specify a non-detach option to enable a parent to maintain a direct link to a new process after startup.

Auxiliary processes are not wholly integrated with the rest of the system in that they cannot communicate using CLAM's protocols. The cost of assigning and revoking CLAM pids to auxiliary processes during their creation/destruction is too high. Instead, each auxiliary process maintains a direct link to its parent—the intermediate process—for the purpose of relaying relevant information to the parent.

Error conditions detected by an auxiliary process are reported to the parent via an intermediate process (with ASCII strings). These error conditions are grouped into two classes: first class errors,

[7] It is permissible for the parent to act as the intermediate process.

which prevent the creation of the new process; and second class errors, which may occur at an auxiliary process once the new process has been successfully created. Such a classification enables a parent to decide when to wait for the reply to a specific spawn request. Timeouts are useful in recovery from second class errors, but are not useful with first class errors (since a new process does not exist). The intermediate process also collects exit status information from auxiliary processes that terminate after creating a new process. Within an intermediate process, a thread waits for status changes in auxiliary processes and reports these to the parent. The exit status alone cannot be used to determine if an auxiliary process has failed; for example, when an *rsh* command is used with a wrong path, the command fails to create a new process and reports the message "Command not found" to its `stderr` file descriptor, with an exit status of 0 (i.e., success). The parent detects such situations by parsing the output string sent by the auxiliary process. With the HTTP-based startup method, errors reported by the HTTP protocol are also detected by parsing the response issued by the server at the parent.

The Spawn RTA message sent by the parent to an intermediate process contains information about the parent and each new process to be created; its format is shown in Figure 12 (with both the "C" and the ASN.1 representation). The parent information is placed at the front of the message. The `type` field specifies the parent's process type. The `ppid`, `oppid`, `prip`, and `port` fields specify the parent's CLAM pid, OS-pid, IP address, and UDP port number, respectively. The `tag` field is the request id number, used by the parent to match replies. The parent information is passed to new processes via command line arguments, enabling every child process to install a process table entry that allows direct communication with the parent.

Other information in the Spawn RTA message specifies the number of new processes to start up, and the startup method. The format of this part of the Spawn RTA message is given by the `SpawnE` data structure shown in Figure 12(a). The `type` field indicates which type (i.e., FS, CP, DS) of process is to be created. The `method` field specifies the creation mechanism (i.e., `rsh`, `fork`, `rexec`, HTTP). The `flags` field specifies configuration options for the new process: the *redirect* option instructs a new process to redirect `stderr` and `stdout` to a file, the *detach* option directs it to detach itself from the parent, and the *display* option instructs it to display its protocol address after initialization. The `path` field contains either the UNIX path or the URL of the executable to be run by the new process. The `dargs` field contains a sequence of null-terminated strings, representing debugging switches (to be passed to the new process). The `args` field is a sequence of null-terminated strings, containing user arguments that must be passed to the new process.

The `methodSpecific` union contains information that depends on the mechanism used to create the new process, as described by the structures `Rsh`, `Rexec`, and `Fork` in Figure 12(a). All the arrays of characters in these structures are represented by null-terminated ASCII strings in the Spawn message. The `host` field contains the name of the machine on which the new process must be started, and the `user` field specifies the process's user-name. The `port` and `pssw` fields are only used with the `rexec` method. These specify the remote port where the *rexec* daemon listens, and the password for accessing the remote host.

The format of the Spawn-Report RTA message is shown in Figure 13. There may be multiple records with the `SpawnRprtEntry` structure in each Spawn-Report RTA message. The `tag` field identifies the startup request for which the rest of the data is being reported; this value is used by Spawn-Report RTA message's handler to integrate the message's information at the parent. The value of the `pid` field depends on the startup method and the result of the startup request. With *rsh* or *rexec*, the `pid` field is set to the OS-pid of the auxiliary process. With *fork*, the OS-pid of the new child process is placed in the `pid` field. With HTTP, if connection to the HTTP server is established, the `pid` field is set to 0. Finally, if a fork operation used to create the auxiliary process or a new process fails, or if the intermediate process is unable to connect and send a HTTP request, the `pid` field is set to -1 and an ASCII error message is placed in the `errstr` field. The `ess` field is divided in two parts, one that specifies the startup method and the other that specifies the state of the `exs` field; the value of the

(a) C structure

```
typedef struct rsh_t {
   char host[ ];   /* Target host */
   char user[ ];   /* -l option for rsh */
} Rsh;

typedef struct rexec_t {
   u_short port;   /* Target rexec port */
   char    host[ ];/* Target host name */
   char    user[ ];/* Login name */
   char    pssw[ ];/* Password */
} Rexec;

typedef struct fork_t {
   char host[ ];   /* Target host name */
} Fork;

typedef struct spwne_t {
   u_short type;   /* Type of Process */
   u_short method; /* Method to be used */
   u_short flags;  /* Flags */
   union mthds_u {
     Rsh   rsh;     /* rsh specific data */
     Rexec rexec;   /* rexec() specific data */
     Fork  fork;    /* fork() specific data */
   } methodSpecific;
   char path[ ];       /* Path to executable */
   char dargs[ ][ ];   /* CLAM-specific args */
   char args[ ][ ];    /* Application arguments */
} SpwnE;

typedef struct spwn_t {
   u_shorttype;   /* Type of Parent Process */
   u_long ppid;   /* Parent CLAM Pid */
   u_long opid;   /* Parent OS Pid */
   u_long prip;   /* Parent IP Address */
   u_shortport;   /* Parent UDP Port */
   u_shorttag;    /* Tag for this request */
   SpwnE  spw[ ]; /* Spawn request entries */
} SpawnMsg;
```

(b) ASN.1

```
SpawnMsg ::= [ APPLICATION 2 ]
SEQUENCE {
   parentProcType ProcType,
   parentPid      Pid,
   parentOSPid    OSPid,
   parentIP       IP,
   parentPort     Port,
   tag            Tag,
   spawnRqsts     SpawnRqsts }
SpawnRqsts ::= SEQUENCE OF
SEQUENCE {
   procType   ProcType,
   method     Method,
   flags      Flag,
   methodSpec CHOICE {
     rsh   Rsh,
     rexec Rexec,
     fork  Fork
   },
   path       IA5String,
   dargs      Args,
   args       Args }
Method ::= ENUMERATED {
   RSH   (0),
   REXEC (1),
   FORK  (2),
   HTTP  (3) }
Rsh ::= SEQUENCE {
   host IA5String,
   user IA5String }
Rexec ::= SEQUENCE {
   port Port,
   host IA5String,
   user IA5String,
   pssw IA5String }
Fork ::= SEQUENCE {
   host IA5String }
Args ::= SEQUENCE OF {
   arg IA5String }
Flag ::= INTEGER ( 0...maxShortInt )
Tag ::= INTEGER ( 0...maxShortInt )
```

Figure 12: *Spawn* RTA Message

(a) C structure

```
typedef struct spwnrprt_t {
   u_short tag; /* Tag of related request */
   u_long  pid; /* OS Pid of new process */
   u_short exs; /* Method + state of exs */
   int     exs; /* Exit status */
   char    errstr[ ]; /* Error description */
   char    chlstr[ ]; /* Child output */
} SpawnRprtEntry;

SpawnRprtEntry SpawnRprtMsg[ ];
```

(b) ASN.1

```
SpawnReportMsg ::= [ APPLICATION 4 ]
SEQUENCE OF SEQUENCE {
   tag           Tag,
   oSPid         OSPid,
   method        Method,
   eState        EState,
   exitStatus    ExitStatus,
   errorString   IA5String,
   childString   IA5String
}

EState ::= ENUMERATED {
   RCVDS   (0),
   TOWFS   (1)
}

ExitStatus ::= INTEGER ( 0...maxLongInt )
```

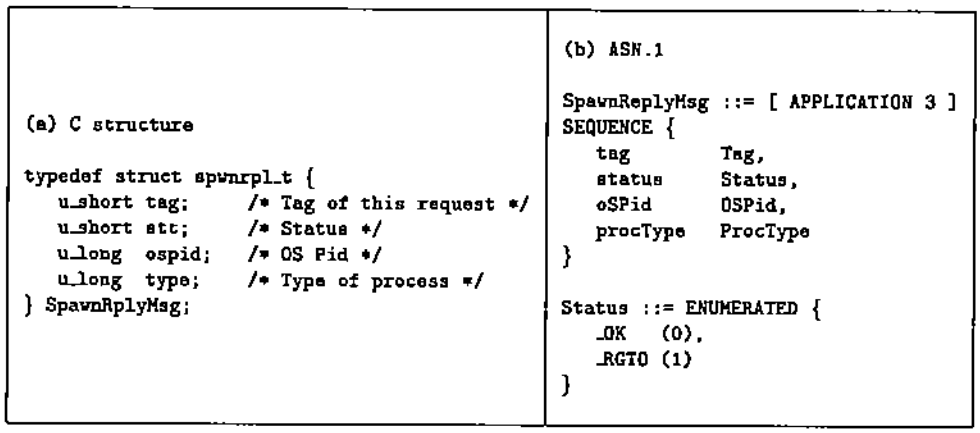
Figure 13: *Spawn-Report* RTA Message

part that specifies the state of exs is either _RCVDS or _TOWFS. The former means that the intermediate process actually received the exit status of the auxiliary process, and this is contained in the exs field. The latter indicates that the intermediate process timed out while waiting for the exit status; in this case exs is invalid. All data collected by the intermediate process from its pipes to its children is placed in the chldstr field of the corresponding request. In addition, all the data collected from TCP/IP connections to HTTP servers that act as startup daemons is placed in the chldstr field of the corresponding request. The errstr and chldstr fields are variable-length, null-terminated ASCII strings.

The Spawn-Reply RTA message is sent by each newly created CLAM process to its parent, communicating the result of a startup operation. The format of the Spawn-Reply RTA message is shown in Figure 14. The tag field facilitates matching replies with their respective *handler records* at a parent process. Each time a parent makes a startup request, a handler record is created and a tag is associated with the request. The handler record contains a tag corresponding to the request and an array of reply structures used to store status information from newly created processes. This status (i.e., stt field) information includes the values _WAIT, _FAIL, _OK, _TOUT, or _RGTO. Here _WAIT means that the parent is still waiting for a new process's Spawn-Reply RTA message, _FAIL means that the process creation request failed, _OK means that the request completed successfully, _TOUT indicates that the request timed out (i.e., the Spawn-Reply RTA message was not received), _RGTO means that a new process was unable to register with its master.[8] The OS-pid (ospid) and the type (type) of the new process are also reported to the parent in the Spawn-Reply RTA message. With this information the parent can create a process table entry for a child, enabling it to communicate directly with each child after initialization.

In addition to the status of each creation request, the handler record also contains a count of the number of new processes that are to be created. Handler records are placed in a global queue where they can be manipulated by incoming Spawn-Reply RTA messages based on tag values. A Spawn-Reply RTA message modifies status information stored at a corresponding handler record when it arrives at a parent. The record is removed from the global queue when the corresponding startup request times out, or when all new processes have reported their status and the parent has received the Spawn-Report

[8]The master is its immediate parent in the CLAM tree-shaped hierarchy.



(a) C structure

```
typedef struct spwnrpl_t {
   u_short tag;      /* Tag of this request */
   u_short stt;      /* Status */
   u_long  ospid;    /* OS Pid */
   u_long  type;     /* Type of process */
} SpawnRplyMsg;
```

(b) ASN.1

```
SpawnReplyMsg ::= [ APPLICATION 3 ]
SEQUENCE {
   tag         Tag,
   status      Status,
   oSPid       OSPid,
   procType    ProcType
}

Status ::= ENUMERATED {
   _OK   (0),
   _RGTO (1)
}
```

Figure 14: *Spawn-Reply* RTA Message

RTA message from the intermediate process. The system does not wait for replies if an intermediate process reports failures during process creation.

In Figure 15 is shown an example with two concurrent startup requests, with messages and the global queue of pending requests. Although message ordering is not explicit in the figure, it is easy to infer from the description given previously. Once a process is created (as a result of a spawn request) it registers for the CLAM session using Register and Install messages. After registration, it exchanges messages to report the status of its creation and to enable direct communication with its parent. One of the requests shown in Figure 15 ($tags = 0, 1$) creates two new processes with the help of a process that is not their master. The second request ($tag = 2$) creates a new process using the process's master as intermediary. A handler is created for each request, and it is identified with the corresponding tags. These handlers are placed in a global queue until requests either complete or timeout.

The process startup interface described here offers some advantages over traditional, single-threaded and centralized implementations. First, as will be shown by our experiments (see Figure 16), CLAM's startup is very efficient for WANs or a set of interconnected LANs. For the creation of multiple processes, it suffices to send spawn requests over a WAN using a single message. In contrast, other message-passing systems require a connection establishment for each new process to be created. CLAM's scheme is also flexible because it offers many methods for creating new processes. Finally, multiple requests can be processed concurrently, due to CLAM's use of threads, non-blocking calls, and auxiliary processes.

3.3.3 Passing Configuration Information to New Processes

Newly created processes are informed of their type (i.e. DS, CP, FS), the process to which they must register (i.e., their master), and the process that requested their creation (i.e., their parent) through command-line arguments. This section describes these command-line arguments along with their format and semantics.

The `-y` switch takes one of the following arguments: `ds`, `cp`, and `fs`, and its function is to inform a CLAM process of its type. The `-R` switch takes two arguments with the following format: `IP-adds:UDP-port`. The `IP-adds` part represents the IP address of the master process, and the `UDP-port` part its UDP port. The `-S` switch is used to provide a CLAM process with information concerning its parent. This switch takes six arguments with the following format:

```
P-type:P-pid:P-opid:IP-addr:UDP-port:TAG
```

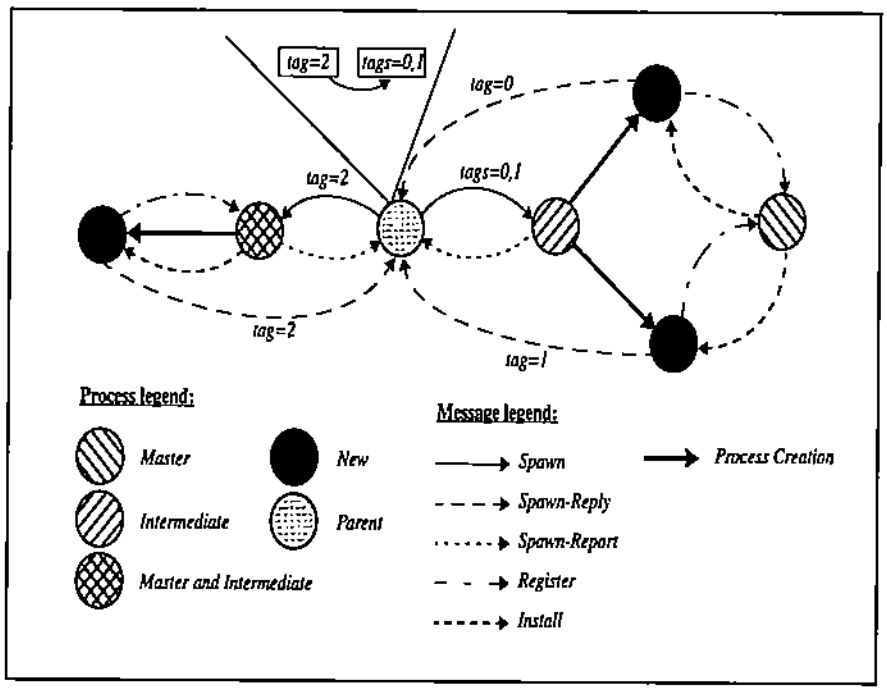


Figure 15: Details of CLAM's Process Spawning

These arguments represent the parent's process type, pid, OS pid, IP address, UDP port, and a tag number, respectively. The tag number corresponds to the spawn request that resulted in the creation of the process accepting the `-S` switch. CLAM processes started manually are not passed the `-S` switch, and they do not send a Spawn-Reply RTA message. The master of these processes assumes the role of the parent process for all the other concerns. The `P-type`, `UDP-port`, and `TAG` numbers are two-octet wide, while `P-pid`, `P-opid`, and `IP-addr` are four-octet wide numbers.

Finally, the `-o` switch informs the new process of some optional settings to be used during initialization and the method that was used for creating it. The method used for the creation of a process is significant because processes created by a HTTP server have to communicate with the server using the HTTP protocol, and not plain ASCII strings like when *rsh* or other startup methods are used. The format of the parameters to this switch is the following: `mthd:flgs`. The `mthd` field is a two-octet wide number and the `flgs` field is four-octet wide.

All the numbers passed as arguments to new processes need to be in network byte order and hexadecimal format. User parameters are passed in the command-line arguments after a colon (:) character, which is used as a delimiter. Debug switches can also be passed through command-line arguments; they are placed before the colon that separates the user-specific arguments and are distinguished by the `-d` switch. The argument to the `-d` switch is generally the name of a function for which tracing information needs to be collected.

The arguments to the `-y` and `-S` switches depend on the initial spawn request. The arguments to the `-R` switch, however, depend on the type of process that is executing the startup request and the type of the new process. If the process executing the Spawn RTA is a DS process, its own IP address and UDP port are used as arguments to the `-R` switch, unless the new process is also a DS process; in this case, the IP address and UDP port of the RDS process are used as arguments to the `-R` switch. If the process executing the Spawn RTA is an FS or a CP process, the IP address and UDP port of

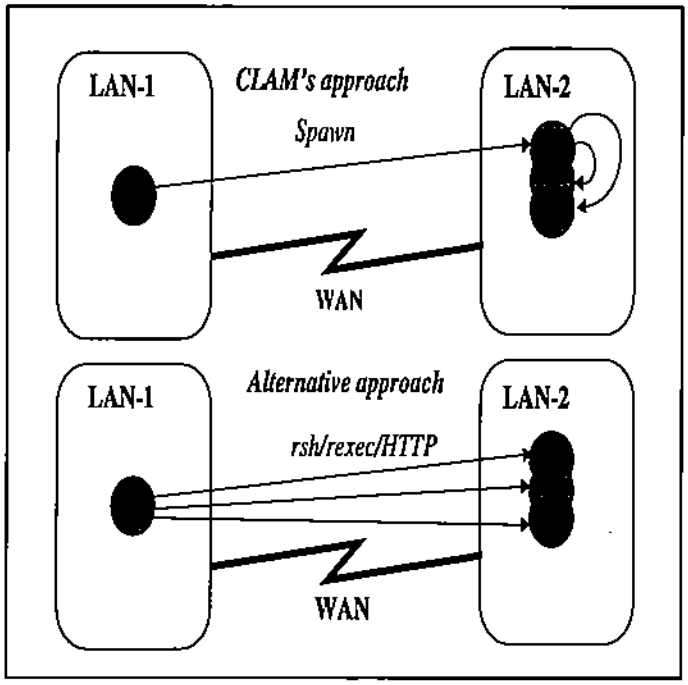


Figure 16: Process Spawning Across Wide Area Networks

their master DS process are used as arguments to the -R switch. Currently, DS processes may not be started up by FS or CP processes because the latter processes may not have addressing information corresponding to the RDS. This, however, may be resolved by having the IP address and UDP port of the RDS installed in the process table of every CLAM process at initialization.

3.4 Process Termination

A CLAM process signs off a distributed session by terminating all its dependents and notifying its master. A process causes the orderly termination of one of its dependents by sending it a *LShutdown* RTA message. Dependant processes notify their master that they are leaving the computation through the *Remove* RTA message. Upon arriving at its destination, the Remove RTA extracts entries specified in its content from the destination's process tables. The format of a Remove RTA is shown in Figure 17(a), represented as a "C" data structure, and in Figure 17(b) as an ASN.1 abstract data type. The LShutdown RTA is not represented because it does not carry any data. Each record within a Remove RTA specifies the CLAM pid and the type of a process whose entry must be removed from the destination's process tables.

The LShutdown RTA causes a CLAM process to start an orderly termination of all its subordinate processes, and then begin its own termination sequence. Figure 18 illustrates a case in which the RDS process sends a LShutdown RTA message to a DS process. Upon receiving this message, the DS process sends LShutdown RTA messages to all its subordinate processes. After sending the these RTAs to all its subordinates, the DS sets a timeout in case one or more dependents fail to respond. After receiving confirmation from all its subordinate processes, the DS proceeds to sign off the distributed session by sending a *Remove* RTA message to its master (i.e., the RDS). The `c_lshtdwn()` CLAM primitive is used to initiate a local lightweight shutdown by any process. The `c_sndlshtdwn()` primitive is used to

```
(a) C structure

typedef struct rmv_t {
   u_long  pid;  /* CLAM Pid */
   u_short type; /* Type of Process */
} RemoveEntry;

RemoveEntry RemoveMsg[ ];
```

```
(b) ASN.1

RemoveMsg ::= [ APPLICATION 6 ]
SEQUENCE OF SEQUENCE {
   pid      Pid,
   procType ProcType
}
```

Figure 17: *Remove* RTA Message

initiate a lightweight shutdown at any other process.

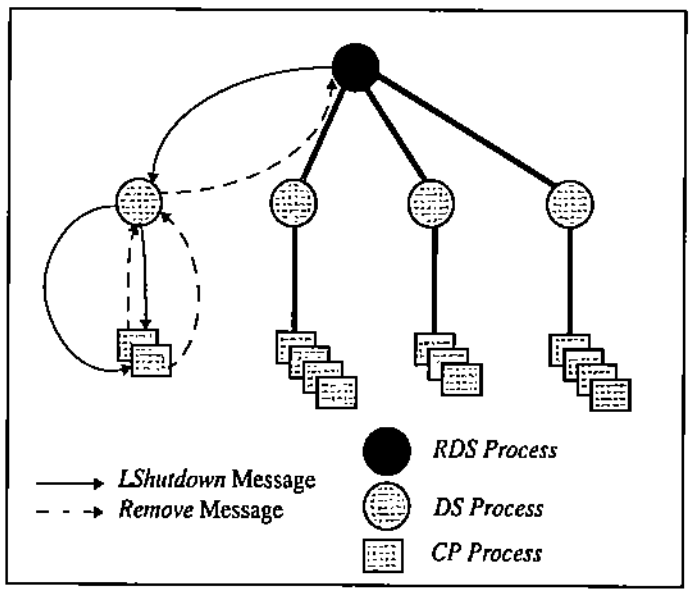


Figure 18: Lightweight Shutdown

The global shutdown (*Shutdown*) RTA message causes all the processes involved in a CLAM session to terminate. When a process receives a Shutdown RTA it terminates all its subordinates and sends a Shutdown RTA to its master. Once all subordinates have confirmed termination or the wait has timed out, the process signs off from its master by sending a *Remove* RTA with its own pid. The Shutdown RTA message is especially useful in handling fatal errors that occur at at any process. The messages involved in a global shutdown initiated by a specific CP can be seen in Figure 19. The `c_shtdwn()` primitive is used to initiate a global shutdown by any CLAM process. The `c_sndshtdwn()` primitive is used to send a Shutdown RTA to a remote process.

The functions that handle the global and lightweight shutdown procedures are non-reentrant and execute only the first time they are called. This prevents potential problems from arising when lightweight or global shutdowns are simultaneously initiated by two different processes. Although the ASN.1 representations of the LShutdown, Shutdown, and Null messages were omitted for brevity, their applications tags are consecutively assigned according to Figure 6 (i.e., these messages are assigned the application tags 7, 8, and 9, respectively).

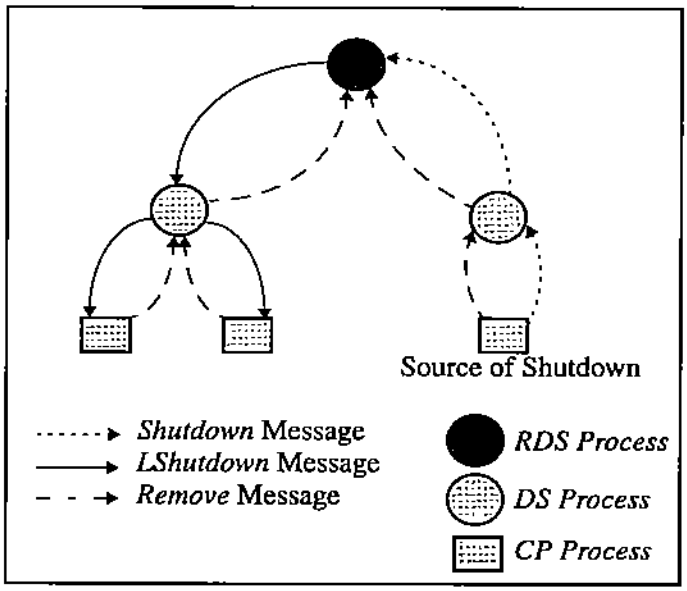


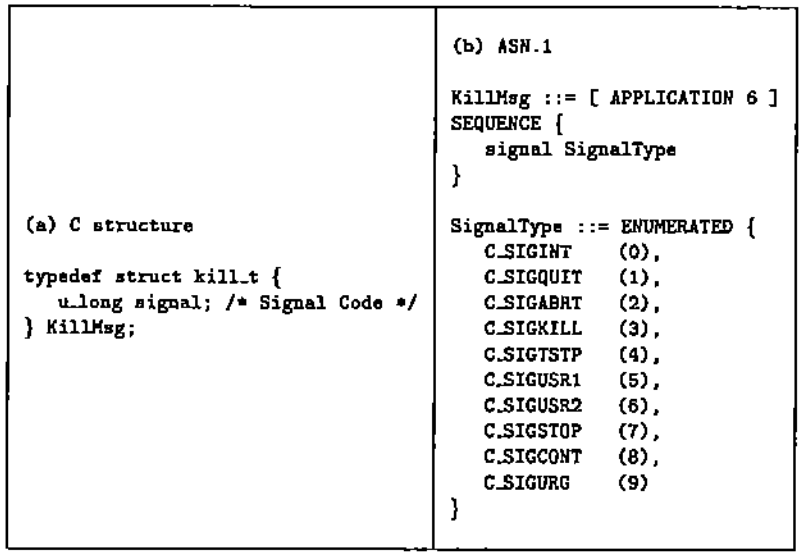
Figure 19: Shutdown

3.5 CLAM Signals

In CLAM, signals can be delivered to remote processes via a *Kill* Remote Thread Activation (RTA). This message contains an unsigned long integer that represents the CLAM code for the signal being delivered (see field `signal` in the "C" data structure shown in Figure 20(a)). Table 3 lists all signals that are currently supported, along with their UNIX equivalents. In Figure 20 is shown the "C" and ASN.1 representations of this message. An internal signal code set is used for portability. CLAM's signal semantic corresponds to that of UNIX System V. A signal can be sent to a CLAM process by invoking CLAM's `c_sndkill()` primitive.

CLAM Code	UNIX Signal
C_SIGINT	SIGINT
C_SIGQUIT	SIGQUIT
C_SIGABRT	SIGABRT
C_SIGKILL	SIGKILL
C_SIGTSTP	SIGTSTP
C_SIGUSR1	SIGUSR1
C_SIGUSR2	SIGUSR2
C_SIGSTOP	SIGSTOP
C_SIGCONT	SIGCONT
C_SIGURG	SIGURG

Table 3: CLAM's Signals



(a) C structure

```
typedef struct kill_t {
   u_long signal; /* Signal Code */
} KillMsg;
```

(b) ASN.1

```
KillMsg ::= [ APPLICATION 6 ]
SEQUENCE {
   signal SignalType
}

SignalType ::= ENUMERATED {
   C_SIGINT    (0),
   C_SIGQUIT   (1),
   C_SIGABRT   (2),
   C_SIGKILL   (3),
   C_SIGTSTP   (4),
   C_SIGUSR1   (5),
   C_SIGUSR2   (6),
   C_SIGSTOP   (7),
   C_SIGCONT   (8),
   C_SIGURG    (9)
}
```

Figure 20: *Kill* RTA Message

4 System Startup based on a Host-file

The user may specify a set of CLAM processes to be started up with the help of an ASCII file. The Extended Backus-Naur Form (EBNF) grammar for this file is shown in Table 4. The first rule for the non-terminal < *line* > is used to specify processes that must be created using the *rexec*, *rsh*, or *fork* methods. The second rule for the non-terminal < *line* > is used to specify processes to be created using the HTTP protocol. The < *password* > and < *port* > non-terminals are used only when the startup mechanism specified is *rexec*. There are three flags that may be set or cleared for each new process: **disp**, **iord**, and **dtch**. The flag **disp** directs the new CLAM process to write its protocol address to its `stdout` descriptor before detaching from its creator. The flag **iord** forces the new process to redirect its `stdout` and `stderr` descriptors to a file. To prevent naming conflicts in systems with NFS mounted files, this file is named using the host on which the process runs and the OS pid of the new process. The **dtch** flag instructs to a new process to detach itself from its parent by closing its `stdout`, `stderr`, and `stdin` descriptors, and forking a new OS process. The < *args* > non-terminal represents an optional list of arguments that the user may pass to a new process as command-line arguments. The < *dargs* > non-terminal represents an optional list of CLAM-specific debug switches that are used to turn on tracing with specific CLAM library functions.

The name of the host-file is passed to the RDS through the `-h` command-line switch when it is started up. If a CLAM process is invoked as an RDS and given this command-line switch, it efficiently starts up all the processes listed in the file. The initialization proceeds as follows. First, the CLAM library parses the host-file and creates a linked list of records containing information on each process to be created. Next, the RDS creates its own children. To avoid kernel resource starvation and excessive network congestion, these processes are started in batches of up to a maximum number specified by an integer constant. Once the children have been started up, the RDS creates a new data structure with the remainder of the records in the initial linked list. The new data structure (shown in Figure 21) has an entry for each new domain to be created. This data structure is intended to facilitate the efficient creation of new domains.

Once the data structure mentioned previously is ready, the RDS starts up the new DS processes.

Grammar Rules		
< *lines* >	::=	{ < *line* > }
< *line* >	::=	< *path* > < *method* > < *type* > [< *password* >] [< *port* >] [< *flags* >] [< *args* >] [< *dargs* >]
< *line* >	::=	< *url* > < *type* > [< *flags* >] [< *args* >] [< *dargs* >]
< *path* >	::=	[[< *user* >@] < *host* > :] < *fs_path* >
< *method* >	::=	**mthd = (rsh | rexec | http | fork)**
< *type* >	::=	**type = (ds | fs | cp | op)**
< *password* >	::=	**pssw =** < *string* >
< *port* >	::=	**port =** < *number* >
< *flags* >	::=	[< *flag* > { < *flag* > }]
< *flag* >	::=	**(disp | iord | dtch) = (on | off)**
< *args* >	::=	**args =** < *arg_list* >
< *dargs* >	::=	**dargs =** < *arg_list* >
< *arg_list* >	::=	< *string* > { , < *string* > }
< *number* >	::=	< *digit* > { < *digit* > }
< *string* >	::=	**ASCII string with no space, tab, newline, or comma**
< *user* >	::=	**Valid login name**
< *host* >	::=	**DNS-compliant host name**
< *fs_path* >	::=	**Valid OS path**
< *digit* >	::=	**0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
< *url* >	::=	**Valid URL with http scheme**

Table 4: Host-file EBNF Grammar

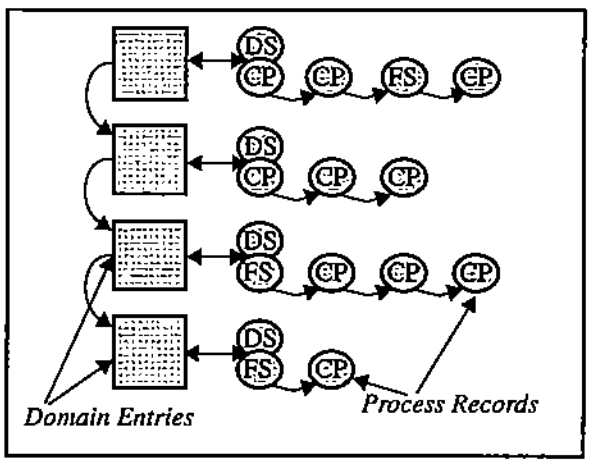


Figure 21: Data Structure Used for Creating New Domains

This task is also performed in batches, for the same reason as described above. A new thread is created for each DS that is started successfully. These threads send Spawn RTA requests to their corresponding DS processes, to have them create their own children. Each thread sends the requests in batches to avoid overloading the DS process and congesting the network. Once the startup procedure is complete, the handlers generated by the various startup operations are analyzed, and errors, if any, are reported. CLAM's startup procedure is specially designed to distribute the load of process creation among the DS processes, and to prevent a buildup of network congestion or kernel resource starvation.

5 Experiments

5.1 Methodology

CLAM uses its PMI intensively during system initialization, and hence, startup time is a critical measure of its performance. To compare CLAM's startup performance with other well-known (single-threaded) message-passing systems, we conducted experiments that measured the amount of time each system took to initialize a set of processes on a workstation cluster. Each process was located on a distinct host. All the experiments were performed on 10 Mbit/sec Ethernet LANs, which were only lightly loaded by other applications.

In each of the systems that we experimented with, startup is initiated by a process that is invoked by the user. This startup *master* reads and parses a host configuration file that specifies the location of each *slave* that must be created, and the creation method. We define *startup time* as the time that elapses between the creation of the master and the time at which all slave processes are ready to begin communication. The startup time is measured by the master, and it includes the time required to read and parse the configuration file. The master is notified of the readiness of a slave through a short (four-byte long) message sent by the slave. For systems that use communication daemons (i.e., PVM, LAM-MPI) we measured both daemon and application startup times.

Because each process is started up upon a distinct host, the number of processes shown on graph axes represents the number of hosts. When the number of processes is one, this process is started up locally. Each observation displayed on a graph is an average over thirty samples. A 90% confidence interval based on the Student-t distribution was obtained for each observation. Interval lengths are negligible with respect to graph scales and are not displayed. We used single CPU workstations (70 MHz SPARCstation 5, SunOS Solaris 5.5) as hosts. In some experiments the master was run on a four-node multiprocessor (50 MHz SPARCstation 20, SunOS Solaris 5.5); the intent is to show how well the different systems exploit shared-memory multiprocessing potential for process initialization. In general, we used executables located on a shared file system, remotely NFS-mounted by each host. To circumvent potential problems of initialization bias due to ARP cache misses and other setup effects, initial observations were discarded.

5.2 Experimental Results

In Figure 22 is shown the startup time (in secs) of daemons, graphed against the number of processes (number of hosts) created, for PVM and LAM-MPI. In Figure 22(a), we observe startup performance when the master is run on a uni-processor (PVM-UP, LAM-MPI-UP) and on a multiprocessor (PVM-MP, LAM-MPI-MP). PVM is clearly able exploit the multiprocessor for concurrency during system initialization; LAM-MPI, however, appears to create daemon processes serially, and thus does not take advantage of the multiprocessor. Because the uniprocessor has a faster clock rate than the multiprocessor (70 MHz vs 50 MHz), the uniprocessor tends to yield smaller times when hosting the master for this small setup. Only up to five processes were used in the experiment corresponding to Figure 22(a), with four remote and one local process; this is because only four additional workstations shared the same

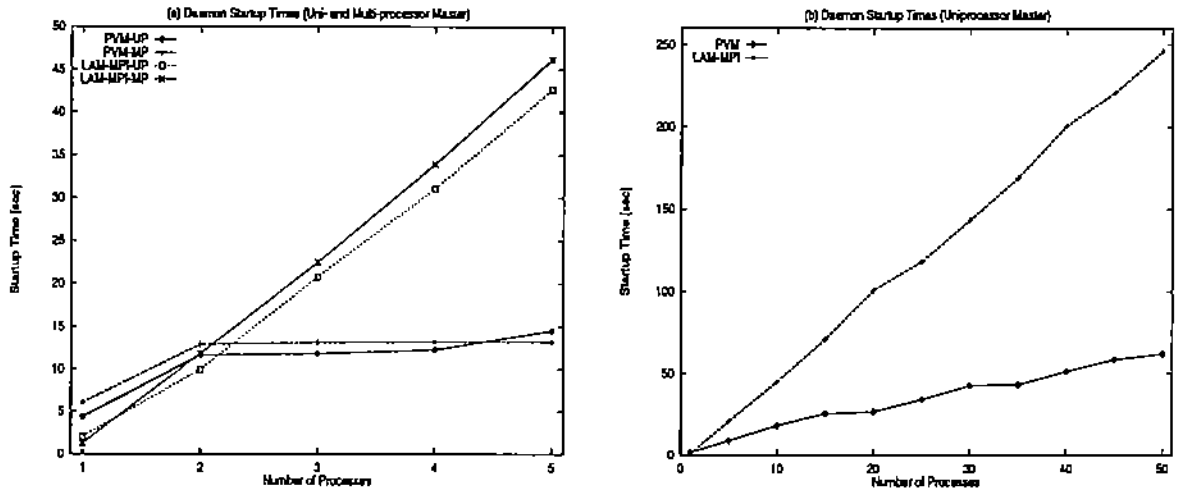


Figure 22: Daemon Startup Times vs Number of Processes

physical network with the multiprocessor. When the number of processes is larger, startup time grows linearly, as can be seen in Figure 22(b). Although startup time increases linearly with the number of processes for both message-passing systems, the increase is rapid with LAM-MPI.

In Figure 23 is shown the application startup time for both daemon-based libraries. Observe that because the daemons are already in-place, the startup time is only a small fraction of the startup time of the daemons. From Figure 23(a) it is apparent that neither system takes advantage of the multiprocessor in starting up application-level processes. In addition, application startup time is seen to grow linearly with the number of processes started up, for both systems. Here, LAM-MPI shows marginally better performance when the number of processes is small. As in the previous graphs, Figure 23(a) shows startup times for both systems when the master is run on a uniprocessor (PVM-UP, LAM-MPI-UP) and on a multiprocessor (PVM-MP, LAM-MPI-MP). In Figure 23(b) is shown startup times for a larger number of application-level processes, and a master process that runs in a uniprocessor host; both systems exhibit similar behavior.

In Figure 24 is shown the results of an experiment that compares the PVM, LAM-MPI, and P4 systems with CLAM, in terms of total startup time vs number of processes created. As shown in Figure 24(a), CLAM tends to offer smaller startup times than the other systems, even for a small number of processes, when the master is run on a uniprocessor. The different CLAM legends identify the different startup methods that CLAM provides: CLAM-RSH corresponds to the remote shell method, CLAM-RXC corresponds to the rexec method, and CLAM-HTTP, corresponds to the HTTP protocol. The CLAM-HTTP graph does not include the startup time for HTTP servers.

A similar performance behavior can be seen in Figure 24(b), where the master is run on a multiprocessor; again, CLAM offers the best overall performance. While P4 and LAM-MPI exhibit graphs with startup times that increase linearly with the number of processors, both PVM and CLAM are fully able to exploit (process-based) concurrency on the multiprocessor during startup. Though PVM shows much larger startup times, the net result for both PVM and CLAM is an almost perfect overlap of the startup of distinct remote processes.

When the number of processes to be started up is large, and the master is run on a uniprocessor, both PVM and CLAM-RSH exhibit similar startup times (see Figure 25). LAM-MPI exhibits sharply increasing times because startup time is large for its daemon. P4 exhibits behavior that is similar to other systems based on remote shell; we were unable, however, to obtain measurements with more

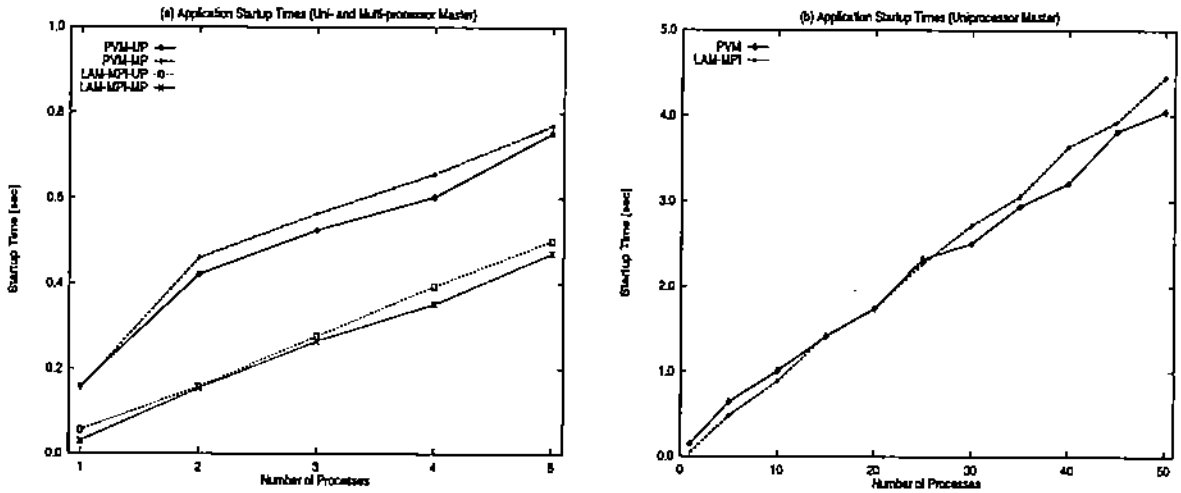


Figure 23: Application Startup Times vs Number of Processes

than 25 processes with P4 because of OS configuration limits on the number of sockets that may be open simultaneously on each process in the system. During these experiments we found that network load played a key role in determining startup times. For example, distributing processes over two different physical networks taxes systems like P4, PVM, and LAM-MPI because startup traffic generated by the process hosting the master cannot be evenly split between the networks. The CLAM system, however, is able to efficiently distribute the network traffic and processing load generated by the startup over two or more networks. The net result is a small startup time, as shown by the CLAM-RSH-DM graph in Figure 25. For this experiment, the startup task was divided equally between two different domain servers (DS) located on two distinct physical networks. This hierarchical arrangement gave an improvement of about 20% in the startup time of a large process set.

As mentioned earlier, we used executables located on a shared file system, remotely mounted by each host with NFS. We attempted to repeat the experiments using executables located on the local disk of each host. We omit these results because they show no significant difference from the measurements with NFS-mounted executables. Although small improvements were observed, these were in the order of 1 to 3% and thus do not warrant special attention. It is possible that with larger executables, slower networks, or a larger number of processes the time required to load executables from a shared file system will become a significant part of the startup time. The testing of such a hypothesis, however, is not in the scope of this work.

System	Daemon [K-bytes]	Application [K-bytes]	Total [K-bytes]
P4	—	—	160
CLAM	—	—	170
LAM-MPI	100	150	250
PVM	800	275	1075

Table 5: Sizes of the Different Executables Used

The sizes of the executable used are shown in Table 5; these sizes may have some influence on startup time, although we expect this influence to be small. Sizes were determined after eliminating

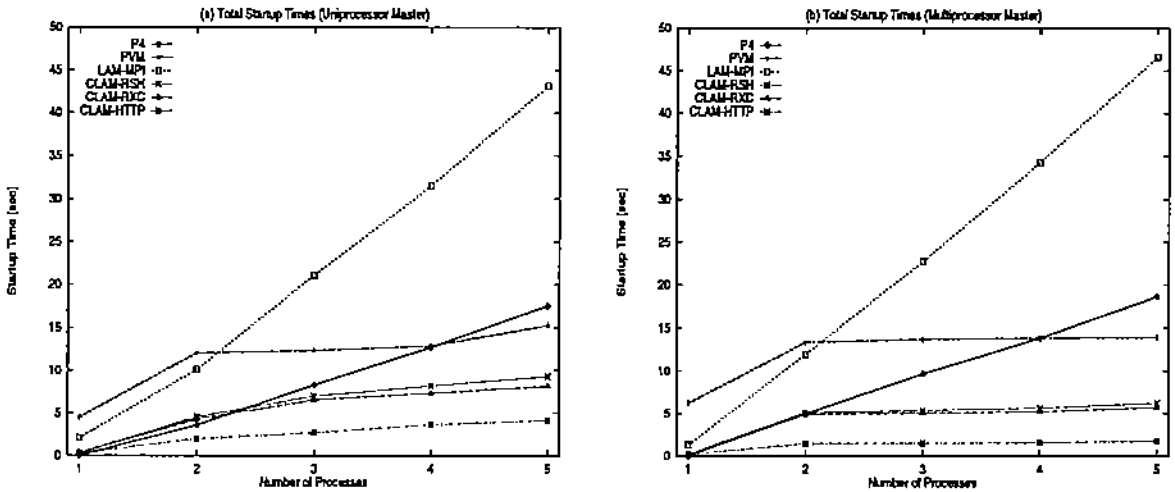


Figure 24: Total Startup Times vs Number of Processes (Detailed View)

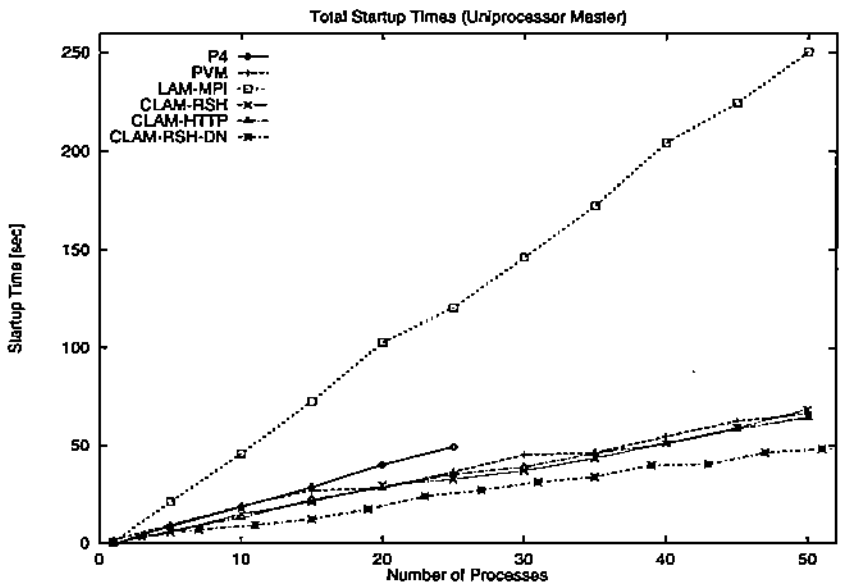


Figure 25: Total Startup Times vs Number of Processes (Expanded View)

debugging and symbol table information inserted by the compiler; all the experiments used the stripped executables. For systems based on communication daemons, we present the sizes of both the application as well as the daemon. The CLAM executable is slightly larger than the P4 executable, but even so, CLAM offers smaller startup times. The LAM-MPI and PVM executables are significantly larger than the CLAM and P4 executables. The CLAM executable used in these experiments did not include its reliable multicast protocol module. However, none of the other systems provide for reliable multicast either.

6 Conclusions

With the help of an efficient implementation of a dynamic process management interface and protocol for distributed system startup, we show how a threads-based protocol can reduce startup time and exploit multiprocessing power. We provide a formal description of the protocol to enable its use and future extensibility. Through a set of experiments designed to compare CLAM's multithreaded startup performance with the performance of other (single-threaded) message-passing systems, we conclude that CLAM's interface is more efficient and exhibits smaller startup costs, with the same—or enhanced—functionality. CLAM's management interface, which uses a pid-based routing scheme to enable communication between processes that do not share a parent-child relationship, exhibits more scalability than that provided by similar systems (e.g., PVM).

This work has also served as a test for the versatility of the Active Messages and Remote Thread Activation interface provided by CLAM's reliable point-to-point protocol module (TRAP) [4]. We conclude that Remote Thread Activations are an ideal transport mechanism for implementing this functionality in a way that is transparent to the application. Further, CLAM's dynamic Process Management Interface (PMI) enlarges the scope of typical distributed applications to include functionality like dynamic load balancing, fault tolerance, and, in general, any functionality that may require periodic runtime assessment and reconfiguration of processes in a distributed session.

Although CLAM's PMI is reasonably mature and well-tested, there are still many areas that can benefit from refinements and enhancements. Kernel-threads, for example, can be exploited during process initialization, to provide more efficient and scalable startup. This is especially true when methods like rsh and rexec are used because the application cannot control blocking (communication) calls that are made internally by these methods. In the current version, we are forced to create new OS-level processes when using these primitives; this enables us to avoid blocking the main process, which handles the startup operation. In addition, kernel-threads will enhance concurrency on shared-memory multiprocessors. We intend to provide this support in a next version. The protocol presented here can be enhanced to include I/O redirection across processes, status queries, and other useful features. Also, reliable multicast can be exploited to decrease network load and enhance performance during initialization.

References

[1] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995.

[2] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[3] A. Skjellum, N. Doss, K. Viswanathan, A. Chowdappa, and P. Bangalore. Extending the Message Passing Interface (MPI). In *1994 Scalable Parallel Libraries Conference*, pages 106–118. IEEE Computer Society Press, October 1994.

[4] Juan Carlos Gomez, Vernon Rego, and V. S. Sunderam. Efficient multithreaded user-space transport for network computing: Design and test of the TRAP protocol. *Journal of Parallel and Distributed Computing*, 40(1):103–117, January 1997.

[5] A. Skjellum, N. Doss, and K. Viswanathan. Inter-communicator Extensions to MPI in the MPIX (MPI eXtension) Library. Technical report, Mississippi State University, APRIL 1994.

[6] K. Al-Saqabi, R. Prouty, D. McNamee, S. Otto, and J. Walpole. Dynamic Load Distribution in MIST. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1997.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[8] R. Butler and E. Lusk. Monitors, Messages, and Clusters: The p4 Parallel Programming System. *Parallel Computing*, 20(4):547–564, April 1994.

[9] Ohio State University. *MPI Primer / Developing With LAM*, November 1996.

[10] L. Revor. *DQS User's Guide*. Argonne National Laboratory / CTD, 9700 South Cass Av. Argonne, IL 60439-4801, 1992.

[11] M. Litzkow, M. Livny, and M. Mutka. Condor—A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 63–71, 1988.

[12] S. Otto. Processor Virtualization and Migration for PVM. In *Proceedings of the 2nd Workshop on Environments and Tools for Parallel Scientific Computing*, pages 66–75, 1994.

[13] J. Gomez, E. Mascarenhas, and V. Rego. The CLAM Approach to Multithreaded Communication on Shared-Memory Multiprocessors: Design and Experiments. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):1–14, January 1998.

[14] L. Wall and R. Schwartz. *Programming Perl*. O'Reilly and Associates, Inc., 1992.

[15] J. Gomez, V. Rego, and V. Sunderam. CLAM: Connectionless, Lightweight, and Multiway Communication Support for Distributed Computing. In *Lecture Notes in Computer Science: Communication and Architectural Support for Network-based Parallel Computing, Springer-Verlag*, pages 227–240, 1997.

[16] E. Mascarenhas and V. Rego. Ariadne: Architecture of a Portable Threads System Supporting Thread Migration. *Software-Practice and Experience*, 26(3):327–357, March 1996.

[17] J. Gomez, V. Rego, and V. Sunderam. On Tailoring Thread Schedules in Protocol Design: Experimental Results. Technical Report TR 96-018, Purdue University, 1996.

[18] J. Gomez and V. Rego. TRAM: A Transaction-oriented Reliable and Multipoint Protocol for Multiway Communication. Report in preparation.

[19] T. von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors.* PhD thesis, University of California at Berkeley, 1993.

[20] D. Wallach, W. Hsieh, K. Johnson, M. Kaashoek, and W. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *Proceedings of the Fifth Symposium on Principles and Practices of Parallel Programming*, pages 217–226, 1995.

[21] CCITT. *Recommendation X.200: Reference Model of Open Systems Interconnection for CCITT Applications.* CCITT, 1984.

[22] CCITT. *Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1).* CCITT, 1988.

[23] B. Kaliski. A Layman's Guide to a Subset of ASN.1, BER, and DER. Technical report, RSA Laboratories, 1993.

[24] CCITT. *Recommendation X.209: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).* CCITT, 1988.

[25] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC-2068, January 1997.

[26] D. Raggett. HTML 3.2 Reference Specification. Technical report, World Wide Web Consortium, January 1997.

[27] University of Illinois at Urbana Champaign National Center for Supercomputer Applications. Common Gateway Interface (CGI). http://hoohoo.ncsa.uiuc.edu/cgi/, 1995.