Distributed System V IPC in Locus: A Design and Implementation Retrospective

Brett D. Fleisch
University of California, Los Angeles

ABSTRACT

This paper describes new interprocess communications facilities that have been added to the Locus system[POPEK 81][WALKER 83]. The facilities improve Locus's interprocess communication repertoire by providing distributed support for three separate subsystems from System V UNIX: *messages, semaphores,* and *shared memory.* Here we describe these subsystems and their integration into in the Locus architecture.

1 Interprocess Communication

Interprocess communication (IPC) has been studied for many years. Historically, the study began in single processor systems. Numerous communication and synchronization mechanisms were developed. Among these, test-and-set, semaphores, shared memory, message passing, and monitors have been used widely. IPC has formed the basis for extending many operating systems into distributed operating systems; communications mechanisms were selected to adapt single machine systems to interact with each other. Typically a special process, called a "network server", provided host-to-host communication and encapsulated the details of the network protocols. In many systems, the interface to the network server used the system's standard IPC calls. This approach appears to be much simpler than building a truly distributed OS kernel. Throughout IPC has been used as an encapsulation or abstraction mechanism.

In distributed systems the development of mechanisms for communication and synchronization has traditionally taken one of two approaches: language oriented or system oriented. This paper will examine the latter approach in the context of a UNIX-based system called Locus. First, however, we must turn our attention to the UNIX system and examine the interprocess communication mechanisms that have been traditionally provided.

© 1986 ACM 0-89791-201-2/86/0800-0386 75¢

1.1 IPC in UNIX

Historically, interprocess communication (IPC) has been supported in UNIX using both *pipes* and *signals.* Pipes[RITCHIE 78] provide a basic facility by which streams of data may be passed between programs. Pipes are a synchronous one-way communication mechanism; the output of one or more programs can be directed to the input of one or more programs through a pipe. A signal, on the other hand, is a mechanism by which one process can inform another about some condition. Thus, pipes are basically a first-in, first-out stream of bytes; signals are a simple asynchronous software interrupt system. Pipes and signals and a transparent distributed implementation have been available in Locus for some time[POPEK 83].

Pipes and signals are not entirely suitable for all forms of communication. While simple and straightforward to use, pipes are limited in functionality and are difficult to use when asynchronous conditions or asynchronous communication must be performed. Signals, on the other hand, may be used for synchronization but not data exchange. Combining pipes and signals to achieve a desirable asynchronous notification and data exchange facility is difficult and inconvenient.

To ameliorate the situation and provide compatibility with System V UNIX, three new facilities have been added to Locus: *messages, semaphores,* and *shared memory. Messages* permit structured communication in discrete packets, providing lightweight packet switching among processes in the network. *Shared memory,* by contrast, provides undisciplined high performance communication; the shared memory subsystem implements *shared segments* using the underlying virtual memory management hardware. *Semaphores* address data access synchronization problems; they can be used to synchronize access to a variety of resources, such as shared memory.

The System V IPC model is a substantial improvement over pipes and signals. One may choose interprocess communication that is more flow controlled and structured; alternately, one may choose IPC that is less flow controlled, less disciplined and exploits the advantages of hardware support. Both approaches go a long way toward improving the rudimentary facilities in standard UNIX systems.

All these facilities are integrated in Locus in a manner entirely compatible with Unix System V IPC. Moreover, Locus supports all but the shared memory facility transparently within an entire network; messages and semaphore sets have a *distributed* implementation.

1.2 System V IPC versus Berkeley IPC

Recently, major UNIX development has been pursued by both AT&T and at the University of California, Berkeley. Berkeley 4.2 UNIX features a version of IPC well understood within academic and industrial circles[LEFFLER 83]. AT&T's System V IPC is less familiar, because fewer academic sites use this version of UNIX. Here we examine the two different models of IPC to gain an understanding of why one would choose one model over the other.

Berkeley's IPC provides two facilities: *virtual circuits* and *datagrams*. These facilities come from traditional networking concepts; we refer the reader to [WATSON 81] for further information. Both virtual circuits and datagrams use the underlying notion of a *socket* which has some of the features of a pipe, some of an TCP/IP connection[ISO 79], and some of an Accent port[RASHID 81]. The socket is the unifying abstraction in this model; communication is directed to sockets.

Berkeley's virtual circuits are more heavily used than its datagram facility. Implemented primarily with TCP/IP as underlying protocol, virtual circuits and the Internet addressing domain are the only implementations fully developed. Predominant use of the virtual circuit facilities using TCP/IP, suggest Berkeley's IPC is best suited for "long haul" environments. Layers of protocol overhead would be wasteful for local area interprocess communication or local communication on the same processor. Nontheless, Berkeley's IPC may be used in any of these situations.

Some have argued that Berkeley's IPC merely provides convenient ARPA TCP/IP style networking similar to facilities long been available on many TOPS-10 and TOPS-20 systems. This tradition of communication has less than desirable performance characteristics; layered protocols add a significant expense to communication. This increases the cost per message transmitted or received. We believe these additional costs make Berkeley IPC most suitable for long-haul communication rather than local area network communication. As an example, when Berkeley 4.2 pipes were built pairing two sockets together, performance of 4.2 pipes was substantially worse than 4.1 pipes in the local case. This is because of the overheads intrinsic in the model. Other performance reports seem to substantiate this claim[GURWITZ 85].

At the other extreme are System V UNIX's facilities. Described briefly in the previous section, the three components of the System V IPC were built for a single system image model of computation. Extending such a model of computation to a distributed environment should reap the benefit of lower costs than a similar conversion for a "long haul" model. Such a system would provide a more lightweight IPC than Berkeley's IPC. Implemented well, this lightweight model would be ideally suited for local area network communication.

1.3 Introduction to the Locus System

Locus is a distributed version of UNIX that provides a superset of UNIX services. Support for the underlying network is almost entirely invisible to users and applications programs. The system supports a very high degree of *network transparency*, that is, it makes the network of machines appear to users and programs as a single computer; machine boundaries are completely hidden during normal operation.

Locus provides a fully transparent file system and facilities for distributed processes. In a Locus network, which may consist of machines of various cpu types, both files and programs may be moved without effect on naming and correct operation. Local operations and remote operations appear the same in Locus. Process creation and migration are permissible and easily controlled by programs and users.

Central to the design of the Locus architecture is the underlying distributed file system. The file system supports a number of high-reliability facilities, including a more robust facility than that of conventional UNIX systems, and support for interprocess communication using pipes. Communication in Locus through *network pipes* operates with exactly the same effect as local pipes. In addition, Locus supports named and unnamed pipes.

When Locus activities must be performed remotely, local system calls are intercepted, a *network message* formatted with necessary arguments and data, and this message transmitted to the remote site. Locus services remote requests by providing a set of lightweight *server processes* which are processes that have no nonprivileged address space. The code and stack of server processes are resident in the operating system nucleus. As remote requests arrive in the form of network messages, these requests are placed in a system queue, and when a server processes finishes an operation, it looks for more work to do in the queue. Each server process serially serves a request. The system is configured with some number of theses processes at initialization time, but that number is automatically and dynamically altered during system execution. The distinction between server processes and application-level user processes is an important one to which we will refer later.

Another aspect of the Locus design relates to a dynamic network environment. In such an environment, network failures and site failures may happen from time to time. These changes in network topology effect the correct operation of protocols in Locus. In particular, any time a site is connected or disconnected from the current *partition* in a network, Locus executes a reconfiguration protocol called *topology change*. The present strategy splits the reconfiguration into two stages: first, a *partition* protocols runs to find fully connected subnetworks; a *merge* protocol runs to merge serveral such subnetworks into a full partition. This protocol detects all site and communications failures and cleans up all effected multisite data structures. Locus assumes a fully connected network, where if host A can talk to host C, and host B can talk to C, then A can talk to B. We will refer to this topology change mechanism later in this paper.

Locus has been operational for over two years on a network of Digital Equipment Corporation VAXs at UCLA. During that time several people expressed frustration with UNIX's rudimentary IPC facilities. The need for more elaborate IPC mechanisms became apparent; work commenced in the summer of 1984 towards this goal. System V IPC compatibility was achieved by mid-summer, but the functions provided were only available on a single site basis. Since then significant new distributed facilities have been added to Locus which we describe here. We begin with an overview of the design.

2 Locus System V IPC Design

We begin our discussion by examining how Locus System V IPC was quite naturally decomposed into two parts and how the overall design was effected by this decomposition. Later sections describe the components in greater detail.

The Locus System V IPC fell naturally into two parts:

1) *naming components:* these are used to maintain the IPC name space. System V Unix names for interprocess communication are not part of the general file system name space. Locus maintains this new name space transparently network wide.

2) *functional components:* these implement the message, semaphore, and shared memory subsystems.

Naming is used by all IPC subsystems; names are used to locate specific IPC communication objects. Those objects will be described in detail later. In Locus these names must be accessible from all sites. The *functional components* comprise the specific subsystems including messages, semaphores, and shared memory. Figure 1 depicts this.

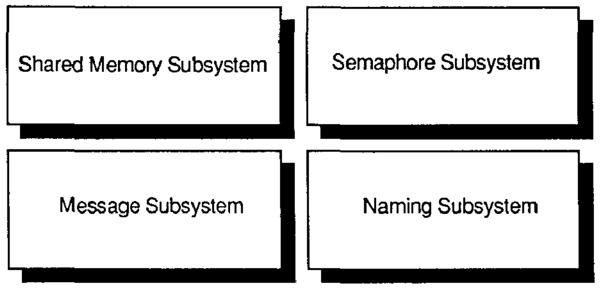

Figure 1: LOCUS System V IPC Components

The separation of naming from the functional components is a natural one. Furthermore, it has effected the choice of implementation strategies. We have observed that the frequency of locating and asserting an IPC name is considerably less than the frequency of use of the functional operations provided by the subsystems. This is often the case in computer systems because names are used typically to refer to objects that will be repeatedly used. This fact permitted us to implement the naming component outside of the kernel in a centralized, yet reliable, manner. Design and implementation were simplified because all of the standard kernel services were available to the naming component. The actual database of IPC names is stored in one special application level server for an entire Locus network. Various IPC system calls will access this database.

Although names are stored outside of the kernel, the IPC subsystems themselves are directly supported by each site's kernel. These functional components consist of a user and storage site component. In the case of local operations, the user site is the storage site. However, when these are distinct, lightweight server processes perform operations at the server site.

Storing the names outside of the kernel raises an issue in communicating values between system calls that require these names and the application IPC name server itself. We have chosen to use the message subsystem itself to provide this form of communication; the IPC name server uses the standard System V message subsystem to communicate values to and from the kernel. Although one system call was added to create a special channel to the kernel, the standard set of message system calls are used for all communication.

2.1 IPC Naming

In UNIX and Locus names are typically found through the file system. For example, all devices are mapped through the file system using the /dev logical directories. Locus is engineered towards providing highly efficient file system operations; a natural consequence of design is that naming is transparent and efficient. Moreover, all names are uniformly found through the file system.

Unfortunately, System V adopted a separate name space for IPC names; IPC names are not part of the file system naming hierarchy. Locus must support this name space to maintain compatibility with software that uses these names. The format of these names is described in the next section.

2.1.1 The Name Format

System V IPC uses two types of names: *keys* and *handles*. *A key* is a 32-bit integer the user selects and associates with a message queue, semaphore set, or shared memory segment. There is one special type of key, with a special value, called the *ipc private* key. If the IPC private key is used, the name of the object is kept private; all other keys reference objects that may be looked up publicly.

Handles operate on objects. Typically when one locates an object using a key, a handle is returned (subject to protections see 1.6). For example, when one tries to locate key "12332" and an invalid handle is returned, one can assume the object does not exist. However, if a valid handle is returned, one can manipulate the underlying object.

2.1.2 Name Service

System V IPC name service allows one to insert, remove, and query keys in the name space. The subsystems described in subsequent sections support these operations through specific system calls. For example, query and insertion are supported with the *msgget(), semget(),* and *shmget()* system calls. The operations *msgctl(), semctl(),* and *shmctl()* provide options for removing an object using a handle. The associated key is deleted at that time.

Most Locus services are kernel-based; however, the implementation of the IPC name database uses a protected application-level process, called the *IPC name server (IPC_NS).* The IPC_NS maintains the name database for all the distributed IPC subsystems, keeping all keys and handles for the subsystems that are distributed. The local kernel maintains this information for the shared memory subsystem.

The IPC_NS is the first application-level server process in Locus that the correctness of services provided by the kernel depends on. Unlike lightweight server processes that are part of the kernel and are used to service incoming network messages, the IPC_NS is a typical application level process with additional functionality. Nonetheless, the IPC_NS is not permitted to tie up kernel resources; it processes requests and commands the kernel to perform certain actions by using system calls.

2.1.3 The Handle Format

As stated earlier, an IPC handle is returned as the result of a lookup operation. Application programmers typically do not inspect the handle. Rather, they present it to the kernel to access an underlying object. Most system calls we will later describe take a handle as their first argument.

The use of handles brings up two issues related to identifying the underlying object to which the handle refers. First, the handle must indicate if the object has been deleted, and another one reallocated, in the same memory location. Thus, a plain vanilla address would be ineffective since a deletion and reallocation could occur between handle uses. This problem was solved in standard System V IPC. Second, if the object is stored at a remote site, it must be possible to detect if the same object is stored there at a later time. This is a problem intrinsic to distributed System V IPC. Recall the site could crash and be revived during handle uses; during that time objects could be reallocated at that site. So, this also says a (site, address) pair would be ineffective as well.

Locus handle format solves these problems. The handle is a 32-bit quantity with three subfields: a *site identifier,* a *bootcount,* and an object *index.* The site identifier is allocated 16 bits, the bootcount 4 bits, and the object index 12 bits. Figure 2 shows the format of a Locus handle. The *site identifier* portion of the handle identifies the home site of the object. Locus IPC objects do not move. Thus, given the handle, one can quickly determine the site where the object resides.

The *bootcount* identifies the current incarnation of the kernel at the storage site of the object. Using the bootcount, Locus can detect if the object's storage site has failed and been revived during intervening handle uses. Locus maintains a running bootcount that is incremented each time the kernel is booted. Handles use 4 bits of this value i.e. in modulo 16 arithmetic. For example, bootcount 161 would be stored as a 1 in a handle (161 mod 16 = 1). If a handle's bootcount was 5, and the current bootcount was computed to be 6, the next time the handle was used Locus would send back an error. The error would indicate the site had crashed and been revived during the meantime. Therefore, it is not necessary to inform holders of handles that a particular storage site has failed. When this happens, the handle is considered invalid.

As in System V Unix, the *index* allows one to determine where, in a fixed size table, the pointer to the object is. When the object is removed, a reuse count in the index slot is incremented so that the index's value is not reused immediately. In this scheme, an index's value is the offset into the table times the reuse count. For example, if there are 50 indices in the table and the zeroth entry has a reuse count of 1, the index value is 50 (50*1), the next allocation of the zeroth slot will use index value 100 (50*2), rather than 50. This continues until a maximum value is reached. Then the reuse index is reset to zero. The value of this scheme should be apparent: the removal of an object and its reallocation in that table index, can be quickly detected. Users with old indices for deleted objects will learn of the table index's reuse upon access.

System V handles can be passed unconstrained, making it hard to detect who possesses handles (object references). For example, it is easy to fork a child process that has the same handle as its parent. Alternately, one could write a handle to a file for others to use. Process migration complicates the issue. Thus, it is nearly impossible to locate all handles. These difficulties indicate process-by-process revocation of handles is infeasible. Consequently, the Locus handle format was designed to make it easy to detect a site failure or object deletion, thus effecting invalidation automatically.

2.2 IPC Protection

Earlier we mentioned that keys were associated with objects. In System V IPC each object, whether that be a message queue, semaphore set, or shared memory segment, is protected independently. The scheme used is similar to that of files, one can specify access by: owner, group, and world.

Protection checks are provided at two times. First, when a key is looked up using the msgget(), semget(), or shmget() system calls protection is checked. Unauthorized access is indicated by an EACCESS error and failure of the system call to return the underlying handle. Second, when using a handle access is rechecked against the protection mask established when the object was created. Thus, users cannot gain access to objects without appropriate permissions. Note however, access to an object that is publicly readable can be obtained by "guessing" handles.

2.3 IPC_NS Kernel Communication

In order to implement much of the functionality described, the IPC_NS and the local kernel must be able to communicate. A new call was added to the Locus system for this purpose called the *ipcnschan()* system call. The call returns the handle of a reserved queue used to communicate with the kernel; this queue is called the *ipcnschan queue.* The ipcnschan queue is a protected, non-removable queue in Locus. It is created when the site is booted and is used to pass values from the kernel to the IPC_NS. The queue accepts only a few types of messages; only the owner (the IPC_NS) may read from this queue using the *msgrcv()* call.

The ability for the IPC_NS to receive messages does little good if it cannot reply. The same handle returned from the *ipcnschan()* call is used by the IPC_NS to communicate replies and requests to other sites. Any message sent with the *msgsnd()* call specifying the handle of the ipcnschan queue is intercepted by the kernel and routed to a destination encoded in the body of the message. In this case, the ipcnschan queue is not used. Clearly, the implementation could have selected another special "dummy" value to send the messages to, but that would waste an extra handle.

The basic operation of the IPC_NS is simple. After a handle to the ipcnschan queue is obtained, the IPC_NS remains in a message receive loop awaiting incoming messages placed on this queue. It uses the standard System V implementation of message receive in awaiting these messages. If a message is received, it is processed, and results are returned to the originating site of the handle. If the message originated locally, Locus will return results to the local process.

Two kernel calls require IPC_NS services: the *msgget()* and *semget()* calls. The Locus kernel implements the msgget() or semget() system call by sending a lookup message to the IPC_NS site and awaiting results. When results are returned either an error condition is raised or a valid handle returned. Two other systems calls, msgctl() and semctl() may be used to remove a handle; in this case the IPC_NS is informed.

Figure 3 shows the flow of control for a typical naming IPC system call. It depicts the sending of network messages (step 1), forwarding of them to a remote site (step 2), unpackaging of them by a server process, conversion of them into an IPC message, and placement of them on the remote ipcnschan queue(step 3). Later, the remote IPC_NS receives the enqueued message and processes it (step 4). Results from the IPC_NS are "sent" to its own ipcnschan queue using the standard System V *msgsnd()* call. Locus intercepts the result message and forwards it in a network message reply to the originator (step 10). Since the originator is suspended in a system call at the local site, the local system call is allowed to proceed when the results of the reply network message arrive (step 11).

When a msgget() or semget() call is issued, an argument may specify that a new queue or semaphore set be created. When the IPC_NS gets the message, it must allocate the object; it then returns a handle to the caller. The policy the IPC_NS uses is that it sends a message back to the using site where the original request was made to allocate the object. Thus, the using site is selected as the storage site for the newly allocated message queue or semaphore set. Therefore, performance is optimized because operations on the object will be local operations.

Figure 3 depicts the additional message request and response (steps 5-9) required to allocate a message queue or semaphore set. The result returned at step 9 must be returned to the client and placed in the IPC_NS database. Since the msgget() and semget() lookup and allocation calls are nearly the same (except for the flag indicating allocation is needed), steps 5-9 must be nested during the standard lookup for the handle to be returned at step 10 and 11.

3 The Message Subsystem

The message subsystem provides a means for one process to communicate with another using discrete packets. Messages are placed in and removed from *queues.* A queue is a repository for messages not yet received by a process.

3.1 Access to Queues

The typical way a user gains access to a queue is to do a *msgget()* system call. Given a key and a flag, this call returns a handle for the queue. The flag may specify that a new queue is to be created or an existing one accessed. If a new queue is to be created, the flag will specifies access rights to the queue as well.

3.2 Sending or Receiving a Message

Once one has access to a queue, a typical operation is to send a message to the queue. The *msgsnd()* system call takes a handle, a pointer, a size, and a flag as arguments. Its purpose is to allow a message to be enqueued for later reception. Given these arguments, *msgsnd()* transfers the *header* of the message to a linked list of headers maintained in system space and the *message body* to another region of system space where it is stored. Lastly, appropriate status fields are updated and if a receiver is waiting for the message just enqueued, it is awakened. The msgsnd() call will suspend awaiting resources (using the internal *sleep()* call) unless the IPC_NOWAIT flag is specified; in that case if resources are not available the error EAGAIN will be returned to the sender.

The *msgrcv()* call is used to receive a message. Msgrcv() takes a handle, a buffer pointer, a size (size of the receive buffer), a type, and a flag as arguments. Msgrcv() will transfer the message from the queue specified by the handle to a buffer in user space. The message type is used to perform a selective receive; it may be used for asynchronous flow control in message receipt. *Msgrcv()* will suspend awaiting a message (or resources) using the internal *sleep()* call, if specified by the flag argument.

The operations described occur if the queue is colocated at the user (message sender or receiver's) site. If the sites are different, the message header and body must be transferred to or from the storage site of the queue. Since IPC messages may be bigger than Locus network messages, this is done by transferring each successive portion in network messages to or from remote server processes. For a *msgsnd()* the servers at the storage site place the contents of the Locus network message into system space until the entire IPC message is completely transferred from the user's site. For a *msgrcv()* the servers remove portions of the IPC message from the system space queue and return the contents in a Locus network message to the using site. In both cases, after the first block of the IPC message is sent or received, sufficient space has been reserved for all subsequent portions to be transferred to the destination.

An issue arises when all resources are exhausted at the storage site during a *msgsnd()* either because all memory in system space is in use, or all headers for messages used. For a *msgrcv()* if the requested message is not found, or for example the queue is empty, it may be necessary to wait until these conditions change. Clearly, what one needs to do is wait, using *sleep()*, as is done in the local case previously described. However, server processes are a limited resource in Locus; it is undesirable to tie up the remote server process sleeping for a resource.

To solve this problem, Locus includes a remote sleep-wakeup facility. Remote sleep wakeup works as follows. When resources are exhausted an error ESLEEP is returned to the using site. The user sends a message to its storage site indicating it desires the resource and is willing to sleep. At that point, the using site's kernel sleeps on a special channel until a message comes in from the storage site indicating the user

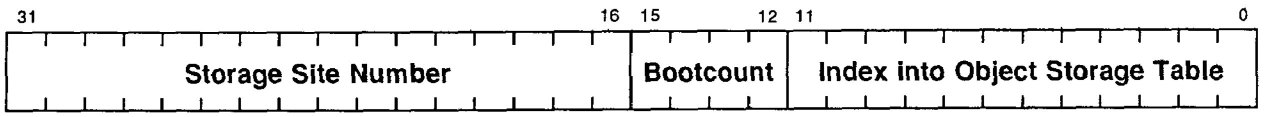

Figure 2: LOCUS Handle Format

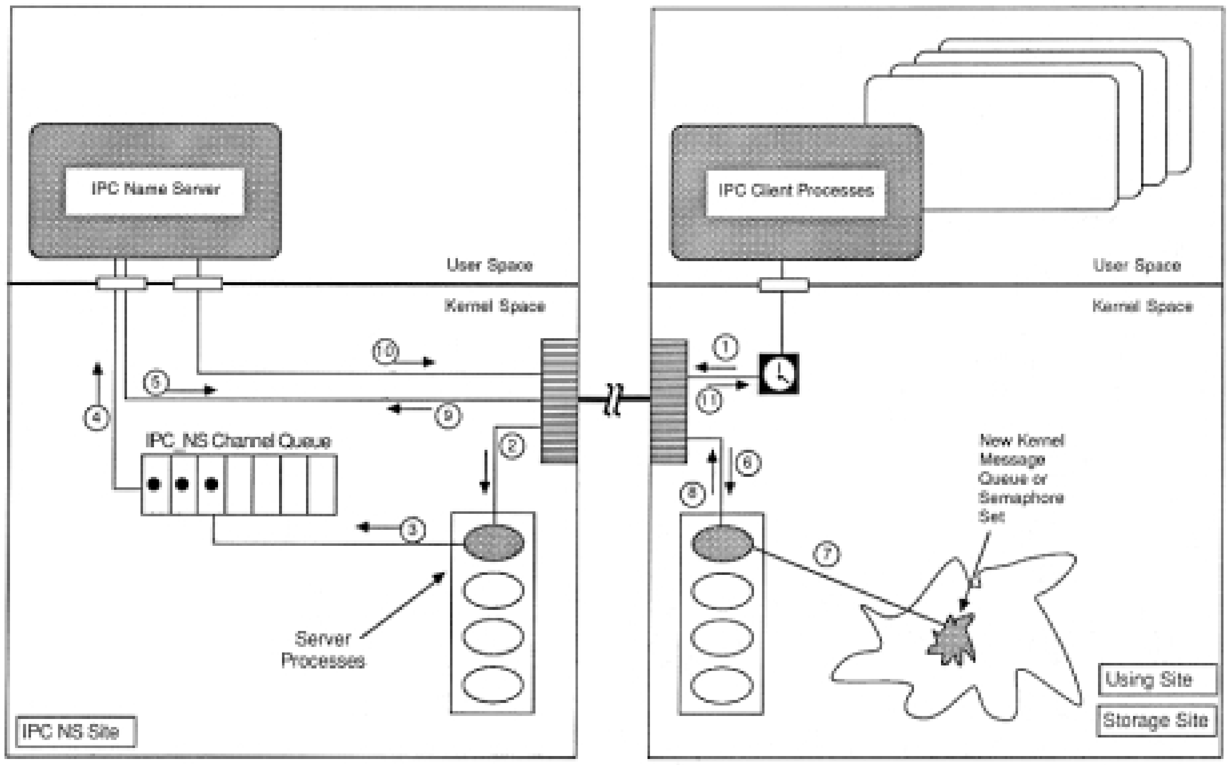

Figure 3: Locus IPC Name Server Allocation Example

should proceed. When that message comes in, the original system call is awakened and proceeds. At that time, the user may retry the original operation. This last message is, in effect, a remote wakeup. If the original system caller, who may be in a remote sleep, is terminated, Locus cancels the sleep at the storage site. This permits proper cleanup of storage structures for process or site failure.

3.3 Other Message Operations

Lastly, the message subsystem supports a *msgctl()* call which implements a variety of operations including: getting the status of the queue, removing a queue and associated key, determining the time the queue was last updated, determining the pid of the last process to update the queue, etc. All these operations are supported by sending a network message to the storage site, when it is remote, and awaiting results. When removing a queue and associated key, the IPC_NS is informed.

4 Semaphore Subsystem

The Locus semaphore subsystem allows one to create logical groups of semaphores called *semaphore sets.* These can be manipulated, named, and updated as one logical unit. Updating a semaphore set is atomic; that is either the operation is performed in its entirety or not at all.

Given a (semaphore) handle, one may perform a number of different *semop()* calls: subject to permissions, a semaphore may read or modified using this call. One particularly useful modification of a semaphore value is to increment or decrement it. Often it is desirable to wait for the semaphore to be decremented to zero or incremented greater than zero. The standard System V semaphore subsystem puts a reader to sleep until the semaphore value changes in this case. This avoids unnecessary polling.

If the same operations, described above, were not performed at the storage site, it would be necessary to package up the arguments to the calls and perform the calls remotely at the storage site. In this using-site/storage-site configuration, all of the typical semaphore operations are handled in just this manner. However, once again it is undesirable to force a server process to sleep waiting for a semaphore value to change. In this distributed case, the remote sleep-wakeup protocol is used.

It is quite possible that operations performed on a semaphore set are interrupted in the middle. For example, a signal could kill the client process in the middle of the *semop()* call. If not handled appropriately, this would leave some of the semaphore values within the semaphore set changed. For atomicity, it is necessary to back out the updates so that changes appear indivisible. The mechanism which is used to implement this is an *undo log*. Each storage site stores an undo log that indicates "how to undo" the semaphore if an application begins an update and is aborted by a signal or site crash. Because semaphore sets do not survive site crashes, this undo log may be stored in memory. The undo log entries are removed when the semaphore is removed; they are process specific. An auxiliary table is stored at the storage site, with the user's pid and the pointer to its in-memory undo log.

If a user with entries in the undo-log terminates, the undo-log must be cleared out. Since termination by signal goes through the exit code, *exit()* has been modified to send a message to the storage site of the semaphore set to clean up this state information. Currently the message is sent to all storage sites where the user has semaphore sets. A special bit-vector is stored in the user structure and kept up-to-date as the user accumulates semaphore sets at different storage sites (by migrating). The bit's position in the vector indicates the site number to send the cleanup message to. On the other hand, if a site fails, it is the responsibility of the topology change routine to update its local undo log. It purges entries from the site or sites that have failed.

Lastly, as in the message subsystem, when one removes a semaphore set, the IPC_NS must be informed so its key is removed from the name space. When Locus removes an IPC handle for a semaphore set, it informs the IPC_NS.

5 Shared Memory Subsystem

This section (for completeness) describes the System V compatible shared memory subsystem by describing the rationale behind what was done in Locus to implement it. Few modifications were made to the standard implementation.

Shared memory gives the user an opportunity to perform undisciplined high performance communication. In this scheme processes in the same proximity (host) of one another can communicate extremely efficiently. A virtue of this type of communication is that communication effects are hard to identify; this totally eliminates kernel overhead in tracking, maintaining, or transmitting values or data structures. Hardware memory management support absorbs most of the overhead.

In a distributed implementation the virtues of the scheme cannot be exploited cheaply. It is difficult to identify *exact* modifications to segment pages and this makes it difficult to come up with an inexpensive transparent implementation. Small changes in a segment page would require transmitting the entire page of the shared segment to other sites. This would be expensive, requiring numerous page-size network messages. Further, hardware memory management would not absorb overhead, as it did in the single-site case. These considerations influenced the decision not to provide transparent shared memory segments between sites.

Along with the implementation of shared segments, is the issue of naming the segments. It was decided that the naming of shared segments be single-site oriented as well; this means that two keys at different sites may be identical. Because of both these factors, once a process begins using the shared memory subsystem the process is forbidden from migrating. Since there is no way to know when a process has released all shared memory segments, once forbidden from migrating, the process will never regain this ability.

Thus, in summary, System V shared memory has not been significantly modified. Rather, controls have been put in place to disallow operations that would cause problems in a Locus network. Nevertheless, we suspect shared memory should be a useful subsystem for co-located Locus clients; the subsystem was easily integrated into Locus with few modifications.

6 Reliability

In distributed system environments complex error conditions can cause unexpected process terminations or site crashes. Nonetheless, reliable operation of the IPC_NS is essential for IPC services. Without an IPC_NS new semaphore sets or messages queues could not be located or created efficiently.

In order to provide reliable name service, it must be possible to detect the termination of the IPC_NS and to restart it without loss of essential information. Since we expect failures to be rare, it is crucial that the IPC_NS operate well in the normal case. In the failure case, it is not crucial that recovery operations be exceptionally high performance.

When the current IPC_NS site fails, it is necessary to select another site to support a new IPC_NS. To conserve kernel space some sites do not provide IPC_NS services, and do not have IPC_NS kernel code compiled in. These sites, which we call *ineligible* must not be selected as IPC_NS sites. Current Locus policies coordinate mounting information when network membership changes. To perform IPC_NS selection, the topology change mechanisms have been modified to gather indications from all sites in the new partition as to whether they are eligible or ineligible to support an IPC_NS. Once this information is gathered, as it must be anyhow for file system mount table coordination, a new IPC_NS is selected as the highest *eligible* site. This provides a relatively low cost means of selecting a new IPC_NS.

Once selected, the IPC_NS starts up with an empty database. To provide truly transparent distributed operation it would be desirable if all (key, handle) pairs not associated with the site which failed i.e. whose storage site was not the site that failed, could be reinserted in the IPC_NS. To provide this reliability, the kernel of each storage site maintains *(key, handle)* pairs for all message queues or semaphore sets stored at that site. This is a critical ingredient to the recovery scheme. By polling all sites, it is possible to fully reconstruct the contents of the IPC_NS from each site's kernel aside from the (key, handle) pairs from the sites which are no longer in the partition. This polling may be done from the application level by starting a small process at each storage site that reads the special /dev/kmem file and transmits the contents over a Locus pipe to the IPC_NS. An advantage of performing key recovery from the application level is that no special purpose kernel code need be written. Indeed the IPC_NS is taking advantage of the high performance pipe code in the Locus kernel.

Although topological changes in the computing environment are one way an IPC_NS may fail, another way for the IPC_NS to fail is for the process to be signalled and that signal not caught. However, the IPC_NS is resilient to all signals and the one signal which cannot be caught is handled with underlying kernel support. If an IPC_NS receives a SIGKILL, a signal which cannot be trapped, the IPC_NS is immediately recreated when standardized *exit()* code is executed. A check is made to see if the process exiting is the IPC_NS. If it is, the kernel informs the recovery master process, which runs on each site, to create another IPC_NS from the replicated directory /etc/ipcns. If a "root" user attempts to create a second IPC_NS on the same site (or different site) the IPC_NS will fail when it performs its *ipcnschan()* call.

One problem that can occur when two or more partitions merge during the network topology change is that each partition may have a separate IPC_NS. Naturally we want only one IPC_NS to be running that has the keys of both of the separate IPC_NSs. One solution to this problem is to select one of the the IPC_NSs as the new IPC_NS and augment its database with all the keys of the others. The current implementation merely kills all IPC_NS's in this merged partition, elects a new IPC_NS and restarts the polling procedure in the new topology. All keys which were in all IPC_NSs will be reconstructed from the kernels of all sites involved. The advantage of this scheme is that all the existing code for election and polling is conveniently reused and no special IPC_NS database merge code need be developed. We felt that this was a reasonable policy to adopt at the time.

One last problem with merging IPC_NSs is that two or more IPC_NSs may have duplicate key values. This situation could occur if two users run the same program in different partitions and the software chose fixed keys values for communication. The situation has no counterpart in a single system environment and indeed there is no "correct" answer to handling this problem. We have implemented a solution of this case by placing only one of the (key, handle) sets in the merged IPC_NS. Existing bindings of keys to handles will be preserved for parties already communicating. New lookups of the key would return the one handle which was stored. However, after a short reevaluation of this issue, we have decided the IPC_NS should accept all duplicate (key, handle) pairs and issue an error EAMBIGUOUS if lookups occur when more than one duplicate (key, handle) pair is in the IPC_NS database.

7 Experiences

The IPC facility took a little over 9 months to implement. The code was produced by the author; his experiences were derived from the RIG[LANTZ 82] project which was a message-based operating system. The Locus kernel had a totally unfamiliar system structure to the author. Design work required several iterations before being adopted. Therefore, a 6-8 month full-time effort could be expected including design work; with an a priori design one could expect to reduce the project to 5-7 months. Figure 4 shows the number of lines of code in the standard System V IPC versus the number of lines of code for Locus distributed transparent System V IPC.

Nondistributed System V IPC Modified for Locus		
Source file	Lines of Code	Single Site System V IPC Sources
msg.c	501	100 lines modified estimate
msginitl	62	(this is a new file)
ipc.c	135	
sem.c	745	200 lines modified estimate
shm.c	607	
Total	2050	
Header File	Lines of Code	
shm.h	67	
sem.h	105	
ipc.h	26	
msg.h	86	
Total	284	

Multisite System V IPC	
Source file	Lines of Code
msg.c	1819
sem.c	1912
ipc.c	438
shm.c	609
ipcns.c	402
gatherkeys.c	230
rkeys.c	87
msginitl	62
Total	5559
Header File	Lines of Code
shm.h	110
sem.h	135
ipc.h	73
msg.h	103
nmipc.h	516
Total	937

code to support:
shared memory throughout kernel
remote sleep-wakeup
cleanup in topology change
reliable IPC_NS

Figure 4: Lines of Code for System V IPC

The surprising amount of time it took to write this application deserves some attention. First, UNIX and Locus suffer from fairly poor tools for kernel debugging. Indeed the only way to debug the kernel when running applications is to take a static core dump. Application-level printfs do little to assist because the underlying buffering may or may not produce the printf before the error occurs. Second, at the time of development, the underlying environment was changing daily; it was hard to keep an experimental kernel up to date with the "current" system. One change in the header of a critical header file would make operating the kernels together impossible because of incompatibilities. An alternate approach was used which was to test the kernel in its "own" partition; in that situation disparities in kernel headers between experimental kernels and production kernels would not cause crashes. Third, because the Locus environment was constantly evolving, the environment's development machines would crash once or twice a day. During those interruptions no code could be written or tested.

Some other experiences merit attention. The IPC_NS communicates to and from the kernel using the standard System V IPC system calls. No modification was made to these call's interface. The technique used is called *bootstrapping*. Our experiences suggest that bootstrapping is very useful. The task of putting the implementation in place was substantially simplified by using the System V IPC itself. Having to reinvent new mechanisms to transmit messages to and from the kernel, with substantially different interfaces would have complicated things enormously. On the other hand, bootstrapping does create restrictions. For instance, for *semget()* to locate a handle when the IPC_NS is co-located at the site of the semget(), requires using the message subsystem. Thus, we have created a dependency of the naming system on the message system and the semaphore system on the message system. For the most part, this is not a problem in Locus.

One of the most important experiences has to do with the structure of the IPC_NS. At first, the IPC_NS seemed to be a serious reliability problem that would require considerable effort to make stable and reliable. Indeed it was obvious from the start that a crash of the IPC_NS would make the IPC system totally unusable. However, the implementation of IPC_NS reliability took an incredibly short amount of time. Most of the time spent on reliability was spent in the design process. The implementation took less than two part-time weeks.

Another issue arises from the standard System V IPC. Since semaphores use the kernel's sleep-wakeup protocol, the fairness of the implementation has to be judged by just how fair the underlying sleep-wakeup is. Unfortunately, UNIX and Locus's sleep-wakeup, whether using the local or remote protocols, is not fair. This could cause one process to unfairly monopolize access to the resource the underlying semaphore protects. This is unfortunate, but could easily be rectified by implementing fairness controls in the wakeup operations.

One policy which may have to be rethought is the one which kills all IPC_NS when partitions merge. Unfortunately, whenever a new site is added to the local net all IPC_NSs are killed. If the site merged into the partition is not going to be the new IPC_NS site, the IPC_NS would have remained at the site where it was before the topology changed. In that case, killing, restarting, and polling is fairly heavy expense. Nontheless, if the site had active IPC entities operating, polling that site and merging IPC_NSs would be essential. Thus, to solve this problem Locus would have to provide mechanisms to distinguish sites with active IPC being merged; a mechanism to distinguish a site being rebooted and merged would also be useful.

We thought the last issue was going to be a serious problem; the System V IPC name space was not part of the file system. The fact that 32-bit keys were used to name a message queue, semaphore set, or shared memory segment did not mesh well with current UNIX or Locus philosophy. Our implementation of an IPC_NS, at first, was meant as a quick means to store values in a general repository. However, later we learned that a separate name space did not pose any intrinsic difficulties we could not solve; it merely added a bit of inelegance to the problem. Our solution of using an application-level name service appears to be the right one, considering we could use all of the facilities of the Locus kernel, including high performance Locus pipes, to implement the IPC_NS and perform key recovery.

8 Conclusions

System V interprocess communication provides a number of ways by which tasks can communicate. Locus has extended these facilities to largely be transparent throughout an entire Locus network. In this way, tasks can use much of these IPC facilities as simply across machine boundaries as within a single machine.

Separating the design into naming and functional components has worked out well. By observing the frequency of name service versus actual subsystem use, the components were implemented to be best suited to this usage pattern. Performance is optimized for the functional components because they are in the kernel itself. Less necessary functions are placed outside the kernel where they may be used without impeding kernel performance or kernel space.

A final consequence of this design is the need to incorporate policies for insuring application-level reliability. This used an IPC_NS selection and update procedure. The selection procedure has little overhead since Locus performs inter-kernel coordination when the topology changes and we have merely added some additional functions to select a new IPC_NS. The second part, polling all kernels, is slightly expensive for failure situations, but has the virtue of requiring little run-time overhead during normal operation.

It is common folklore in distributed systems that a central server model is not a reliable way to structure a computer system. Typically when reliability is needed for this organization a duplicate, mirror, or "hot" backup process is also available at another site. In our case, information is redundantly stored in each kernel and the kernel provides mechanisms for reliability. Reliability is obtained without a duplicate or mirror process and the added overhead of checkpoints and synchronization between the two. The disadvantage of providing reliability in this manner is the amount of time to perform recovery. Nonetheless, this application's modest recovery time appears acceptable.

9 Acknowledgements

Jerry Popek read numerous drafts and revisions of this paper. He also supervised work on issues of System V IPC design along with Bruce Walker who implemented the remote sleep-wakeup. Matt Weinstein, Scott Spetka, and Alan Downing provided comments on earlier drafts of this paper; Weinstein also generously helped in preparation of the figures. Monique Benarrosh proofread the submission draft. Steve Kiser assisted at numerous times during the implementation of this work.

10 References

[GURWITZ 85] Gurwitz, R., An Informal Critique of 4.2BSD, Unpublished Memo, BBN Laboratories, 1985.

[ISO 79] Reference Model of Open System Interconnection, ISO/TC 97 SC16 N 227, August, 1979.

[LANTZ 82] Lantz, K. A., Gradischnig, K. D., Feldman, J.A., Rashid, R.F., Rochester's Intelligent Gateway, *Computer*, October, 1981, pp.54-68.

[LEFFLER 83] Leffler, S. J., Fabry, R. S., Joy, W. N., A 4.2BSD Interprocess Communication Primer, Technical Report, Department of Computer Science and Electrical Engineering, University of California, Berkeley, February 1983.

[POPEK 81] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G. and Thiel, G., LOCUS: A Network Transparent, High Reliability Distributed System, *Proceedings of the Eighth Symposium on Operating System Principles*, Published as SIGOPS Operating Systems Review, Vol. 15, No. 5, October, 1983.

[RASHID 81] Rashid, R.F., Robertson, G.R., Accent: A Communication Oriented Operating System, *Proceedings of the Eighth Symposium on Operating System Principles*, Published as SIGOPS Operating Systems Review, Vol. 15, No. 5, December, 1981.

[RITCHIE 78] Ritchie, D., Thompson, K., The UNIX Timesharing System, *Bell System Technical Journal*, vol. 57, no. 6, part 2, July-August, 1978.

[WALKER 83] Walker, B., Popek, G., English, R., Kline, C., Thiel, G., The LOCUS Distributed Operating System, *Proceedings of the Ninth Symposium on Operating System Principles*, Published as SIGOPS Operating Systems Review, Vol. 17, No. 5, October, 1983.

[WATSON 81] Watson, R. W., Hierarchy, in *Distributed Systems - Architecture and Implementation: An Advanced Course*, Lecture Notes in Computer Science, Springer Verlag, 1981, pp. 109-118.