An OSF/1 Unix for Massively Parallel Multicomputers

Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, & Durriya Netterwala
– OSF Research Institute and Locus Computing Corporation

ABSTRACT

This paper describes the architecture and implementation of a version of the OSF/1 operating system designed to run on multicomputer hardware platforms. The multicomputer hardware platforms targeted can consist of hundreds or even thousands of individual nodes, where each node consists of one or more processors.

The multicomputer version of OSF/1 (called OSF/1 AD TNC) is built on the Mach 3.0 Microkernel and the OSF/1 MK Single Server. These have been modified to run in the multicomputer environment and provide a view of the hardware that looks like a conventional but massively scaled up shared memory multiprocessor. The operating system presents this notion of a Single System Image by building Unix functionality on top of base Mach services running on each node in the multicomputer.

The focus of this paper is on the particular enhancements made to standard OSF/1 functionality to operate in a multicomputer environment without incurring system bottlenecks. These include a new distributed file system, a distributed implementation of sockets, and enhancements to process management functionality to support remote processing and load leveling. Extensions to the operating system interface to allow users to take advantage of the parallelism of the multicomputer hardware are also discussed.

Introduction

Historically, hardware vendors have two strategies that can be used individually or in combination to increase performance: (a) increase processor performance and (b) utilize parallelism.

The parallelism architecture that has received the most attention in the recent past has been SMP (Symmetric MultiProcessing), which is a shared-memory, UMA (Uniform Memory Access) architecture. The SMP architecture is attractive because the operating system can be multithreaded to run on multiple processors. Because of the global shared memory, user processes can simply run on any processor that is available. Unfortunately, SMP systems do not scale past tens of processors.

Another type of parallel processing is MPP (Massively Parallel Processing). MPP computers, also known as *multicomputers*, can consist of hundreds or even thousands of *nodes* (each node consisting of one or more processors) connected via a high-speed interconnect. A multicomputer in which each node is able to access local memory only is known as a NORMA (NO Remote Memory Access) computer. A multicomputer with the additional ability to access memory in other nodes is known as a NUMA (Non Uniform Memory Access) computer. Multicomputers of both types are currently being built and marketed by several manufacturers.

In this paper, we are primarily addressing the concerns of NORMA multicomputer systems. Many of the software concepts described in this paper would apply to NUMA multicomputer systems, but the reference implementation is for a NORMA multicomputer system. For the rest of this paper, when "MPP systems" or "multicomputers" are referenced, it is "NORMA multicomputer systems" that are actually being referred to.

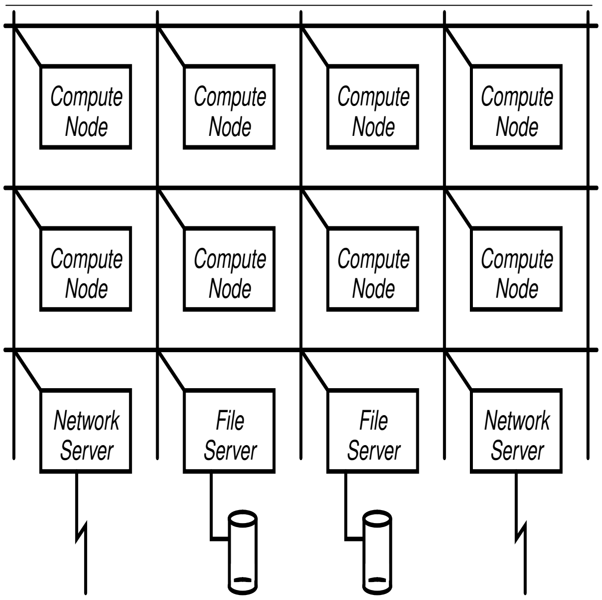

Figure 1: Multicomputer Architecture

In a typical multicomputer system (see Figure 1), the nodes of the multicomputer are divided into three groups: nodes used for input/output and connectivity (I/O nodes or file server nodes); nodes dedicated to parallel applications (compute nodes); and nodes for interactive use (service nodes). The number of nodes in each division varies from system to system – indeed, the division between service nodes and compute nodes may vary depending upon the time of day.

Although the hardware of multicomputers is inherently scalable, a problem has been the provision of a scalable, distributed operating system. While proprietary operating systems for multicomputers have been delivered in the past, current wisdom dictates the necessity for an open operating system and Application Programming Interface (API). Hence, a version of OSF/1 that runs on multicomputers has been developed.

This paper describes OSF/1 AD TNC[1], the multicomputer version of OSF/1 that makes a multicomputer appear as a massively scaled up SMP computer, while avoiding OS bottlenecks. OSF/1 AD TNC was produced by the Open Software Foundation and Locus Computing Corporation, and is based on an enhanced version of the OSF/1 MK Single Server, which is itself based on the OSF/1 operating system and the Mach 3.0 microkernel.

After providing background information on technology incorporated into OSF/1 AD TNC, the following sections describe its goals, architecture, and detailed design. The paper concludes with sections on related work, implementation status, lessons learned, and planned future enhancements.

Background

Mach 3.0 Microkernel

Mach provides a flexible execution environment for both system and user applications. It exposes the management of CPU, communication, virtual memory, and secondary storage resources in a manner that allows system applications to make effective use of these resources. The important features of Mach are:

- *Task and Thread Management*: Mach supports the task and thread abstractions for execution management. A task is a passive resource abstraction, consisting of an address space and communication access to system and server facilities. Computation within a task is performed by one or more threads that fully share the address space and all other resources of the task. Thread scheduling to processors is controlled by the Mach kernel, and threads can execute in parallel on shared memory multiprocessors. Both timesharing and fixed priority policies are supported.
- *Interprocess Communication*: Mach provides interprocess communication via ports and messages. This is actually inter-task communication, but is known as IPC for historical reasons. Ports are protected communication endpoints; only Mach tasks with appropriate capabilities (known as port rights) may send or receive messages on a port (at most one task may receive, but multiple tasks may send). All services, resources, and facilities exported by the Mach kernel and servers are represented by ports. For example, Mach tasks and threads are manipulated by sending messages to ports that represent them.
- *Memory Object Management*: The address space of a Mach task is represented as a collection of mappings from addresses to offsets within memory objects. A Mach memory object represents a single source of memory (e.g., the file from which an executing program was loaded). The kernel manages physical memory as a cache of memory object contents; access to the actual memory (i.e., backing storage) is via a Mach port to which messages can be sent containing data or requesting that it be supplied [Young 87]. This allows memory objects to be implemented by user-state programs such as a file system server or database application.
- *System Call Redirection*: The Mach kernel allows system call traps to be handled in user mode by code executing in the same task. This supports binary compatibility with existing applications without inserting additional code (e.g., from Unix) into the kernel. Similar facilities are provided for redirecting exceptions to user mode handlers [Black 88].
- *Device Support*: The Mach kernel provides low-level device support [Forin 91]. Each device is represented by a port to which messages can be sent to transfer data and control the device. The request and reply messages for the read and write operations are exported as distinct interfaces, supporting both synchronous and asynchronous I/O interactions.

Mach and Multicomputers

Mach supports multicomputer systems by transparently extending communication and memory management services across the multicomputer's interconnection network. Mach's communication mechanisms are location transparent, and can be extended across local area networks by user mode communication servers [Sansom 86]. By comparison to local area networks, well-designed multicomputer

[1] OSF/1 AD TNC stands for OSF/1 with Advanced Development (AD) extensions from OSF Research Institute and Transparent Network Computing (TNC) extensions from Locus Computing Corp.

interconnects have order of magnitude or more increases in bandwidth and reduction of hardware and software access latency [Intel 92]. The multicomputer extension of Mach IPC [Barrera 91] (known as NORMA IPC) is implemented in the Mach kernel to take advantage of the interconnect performance, and allow direct communication among Mach kernels on different multicomputer nodes (each node runs a separate Mach kernel). This IPC extension is completely transparent to Mach tasks; the locations of the communicating tasks (same or different nodes) does not affect the behavior.

The memory management extensions for multicomputers are motivated by practical concerns. Mach allows memory object managers to support object access by more than one kernel and manage the resulting distributed consistency. As a result, NORMA IPC is sufficient to support multicomputer systems implementation, but in practice most memory object managers do not implement multikernel memory object consistency, usually for complexity reasons. Such managers do not work correctly on a multicomputer, and situations in which they malfunction are more common than one might expect. Although actual use of shared memory is quite rare, lazy evaluation of inherited memory (e.g., across a `fork()`) causes multiple nodes to access the same memory when the child task is created on a node other than the parent's.

To support the use of arbitrary Mach memory object managers on a multicomputer system, the XMM (eXtended Memory Management) subsystem has been added to the Mach kernel [Barrera 93]. Like NORMA IPC, XMM is part of every Mach kernel on a multicomputer; its implementation of distributed shared memory makes the collection of Mach kernels on a multicomputer behave as if they were a single kernel when communicating with memory object managers. This removes the complexity of distributed shared memory functionality from the memory object managers. XMM also includes support for copy on reference between nodes for lazy evaluation of memory inherited between tasks on different nodes; this support is used only for memory managed by memory management interfaces, as NORMA IPC does not participate in copy on reference.

OSF/1 Operating System

The Open Software Foundation's OSF/1 is a standards-compliant open operating system that incorporates advanced features while providing compatibility with industry standards and support for existing applications. The advanced features include support for shared memory multiprocessors, B1-level security, logical volume management, multithreaded applications, dynamic system configuration via loadable kernel modules, internationalization and localization (foreign language and culture support). Compatibility is provided to interface specifications from the System V and POSIX collections, as well as for the 4.3BSD programming environment. Further details can be found in [OSF 90] [OReilly 91]

OSF/1 is based on the Mach 2.5 operating system from Carnegie Mellon University. Both are integrated or monolithic kernel systems that implement the majority of system functionality in the operating system kernel (but not all, for example OSF/1's program loader runs in user space). Their kernels contain the core Mach technology and additional Unix functionality. OSF/1 has significantly upgraded the 4.3BSD portion of Mach 2.5 to implement advanced features and obtain compliance with standards.

OSF/1 MK Single Server

OSF/1 AD TNC is based on the OSF/1 MK Single Server. The OSF/1 MK Single server was created by dividing the OSF/1 monolithic kernel into the Mach 3.0 kernel and the OSF/1 MK Single Server. This was achieved by replacing OSF/1's Mach 2.5 internals with Mach 3.0, and adding a layer of compatibility code to allow the rest of OSF/1 to execute as a user-mode server (the OSF/1 MK server). Among the features of the compatibility code are a threads library that provides lightweight user threads on top of Mach's kernel threads, and an emulation library that implements some system functionality in application address spaces. This work was based on an earlier conversion effort at CMU involving the Mach 2.5 system [Golub 90]. The resulting OSF/1 MK server is pagable, preemptible, and multithreaded (leveraging OSF/1's support for symmetric multiprocessors).

Using OSF/1 MK Single Server as the base for OSF/1 AD TNC simplifies the transition to a multicomputer by expressing all of OSF/1's features in terms of Mach's location transparent kernel abstractions.

The OSF/1 MK system is based on a high level mapping of the Unix process model to Mach abstractions. The server is represented to the process by a single port (used for all service requests), and this port serves to identify the requesting process to the server. A system call trap executed by the application is redirected to an emulation library in the application's address space. This library converts most system calls to service request messages and sends them to the server. A shared memory window between the server and each application is used to optimize some data transfers in a fashion similar to (but more primitive than) LRPC [Bershad 90].

Goals

The primary goal of OSF/1 AD TNC was to build a distributed operating system that enables users to take advantage of multicomputer hardware, while retaining full OSF/1 (and thus Unix) semantics. This means that:

- There is a single file name space that spans all the nodes, and access to all files retains *single system image* semantics.
- Scalability bottlenecks are avoided in the operating system, by distributing control of the following OS subsystems:
  - file system
  - socket protocol stacks
  - process management
- Processes executing on any node of the multicomputer have access to all resources and facilities of OSF/1, as if they were executing on an SMP architecture (including maintaining shared memory between nodes).
- Existing applications can take advantage of all the computational resources of the multicomputer via automatic load leveling, while at the same time specially parallelized programs can be explicit in their use of multiple nodes.
- Full compatibility with OSF/1 1.0 commands and libraries is maintained (implying compatibility with XPG/3, POSIX 1003.1, SVID Issue 2, and with other standards).

The initial performance goal is to approximate the performance of the OSF/1 MK system, after taking into account inherent discrepancies due to the multicomputer architecture (such as the fact that peripherals are remote). This goal will be refined and expanded as experience and measurements are acquired.

Architecture

The operating system architecture is derived directly from the target hardware architecture, in which large numbers of nodes are connected via a high-speed interconnect, and some subset of the nodes have peripheral devices attached. The OS must allow all nodes and devices to be efficiently utilized, and in particular, must distribute OS functionality in order to avoid bottlenecks.

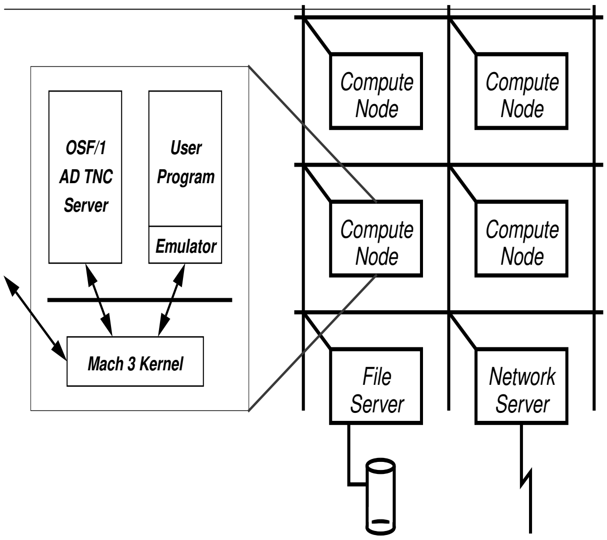

Figure 2: OSF/1 AD TNC Architecture

This leads to the following architectural model (see Figure 2):

- The Mach microkernel runs on all nodes of the multicomputer, providing generic task/thread management, memory management, communication services, and device access.
- Specialized user-space servers implement Unix functionality, such as file service, process management, and networking.
- Disk and networking devices are managed by servers typically co-located on the same node with the device (although co-location is not mandatory).
- Process management functionality is distributed across most or all nodes on which application processes are run.
- An *emulator* (the previously mentioned emulation library) [Golub 90] [Julin 91] is available in each process' address space to provide some Unix functionality and perform system call-to-message conversion. It also contains a thread to receive callback messages from servers in support of interruptible system calls and file caching.

  Process management system calls that require client-server interaction are converted to messages to the process management server, which is normally co-located on the same node as the process. File management system calls that require client-server interaction are converted to messages to the file server that is managing the particular file being manipulated.

This model served as the basis for implementing OSF/1 AD TNC, as described in the following sections.

Detailed Design

In order to meet the above goals, the base OSF/1 Unix code was enhanced in the following ways:

- The file system was modified to support the integration of multiple file servers into a coherent whole, providing a single file system name space, location transparency, and remote device handling. Typical file system-related system calls require at most one client-server interaction. Extensive caching is used to allow many frequently used operations to require no client-server interaction.
- Control of socket protocol stacks is distributed across multiple nodes. This was done in a way to minimally impact the existing code in the protocol stacks.
- Process management is distributed across all nodes of the multicomputer. Many system

calls are completely serviced by the server on the node local to the issuing process without necessitating internode communication. However, system calls affecting process relationships spanning multiple nodes (for example, signaling a process on another node) is conducted in a way transparent to all user processes.

Simple primitives were added to allow programs to explicitly use multiple nodes. Automatic load leveling to utilize all the nodes of the multicomputer was also added.

File System Enhancements

Within the multicomputer, certain nodes are used as file servers. Each file server provides file service for one or more file systems (partitions). Mount operations are used to assemble multiple file systems across multiple file servers into a single file name space. Access to all files within the name space, including to remote devices, is location transparent. Since devices can reside anywhere in the name space, device naming is separate from device location.

To achieve high levels of performance, extensive distributed caching of file data is done within the context of the user process, so that some frequently used system calls do not require client-server interaction. This caching makes extensive use of the Mach memory object abstraction. For file-related system calls that cannot be handled by the distributed cache, typically at most one client-server interaction is required.

As the file server for a file and the process management server for a process may be on separate nodes, a *Credentials Service* was implemented to allow state to be exchanged between file servers and process managers. The fact that file-related system calls may be processed on a node other than the process management node also necessitated a special design for file-related system calls interrupted by signals.

File Servers

File servers are user-space servers that provide access to UFS file systems (the NFS Support section discusses NFS access). Each node may have *at most one* file server, and all of a node's (file system) devices are managed by *exactly one* file server. Typically, the file server for a node's devices will run on that node, but it may also run on a different node (because of Mach's network transparent device interface).

Distributed File Name Space

Construction of the file name space is based on the mount model, whereby a file system (partition) is mounted on top of a directory. A file system and the directory that it's mounted on need not be co-located on the same node, thus allowing file systems managed by multiple file server nodes to be assembled into a single file name space. The resulting behavior is exactly as expected from a standard OSF/1 system where a (local) file system is mounted on a (local) directory. In other words, *single system image* semantics are maintained. An example file name space is shown in Figure 3.

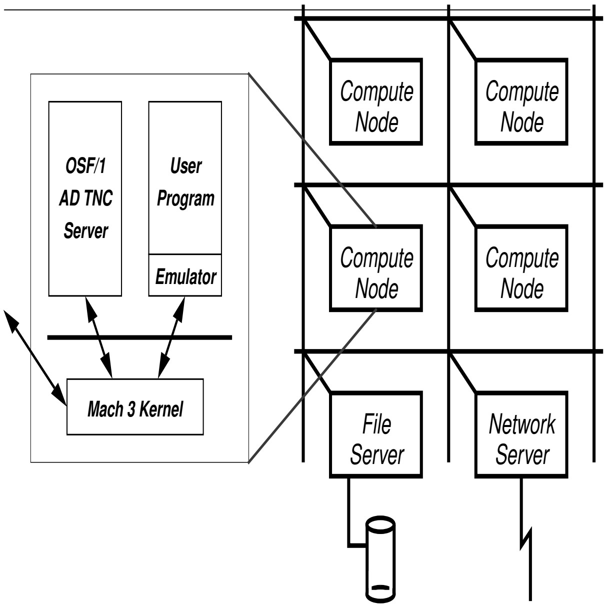

Figure 3: File Name Space

As in standard OSF/1, file systems are identified by device special files in the name space. However, in a multicomputer environment another piece of information is required: the *node* to which the device partition is attached. This is specified by a node number field in the special file's inode. It is set using a new command `rmknod` that is identical to `mknod`, but accepts the node number as an additional argument. This node number is used by the `mount` operation to construct remote mounts.

A `remote mount` is represented by a pair of `mount` structures, as shown in Figure 4. A local structure resides in the file server managing the covered directory and a remote structure resides in the file server for the mounted file system. The local mount structure contains a flag indicating it represents a remote mount, and a send right to the port associated with the remote mount structure. The real information for the mount is contained in the remote mount structure. It also contains a send right to the port associated with the local mount structure in order to support pathname traversal of "..".

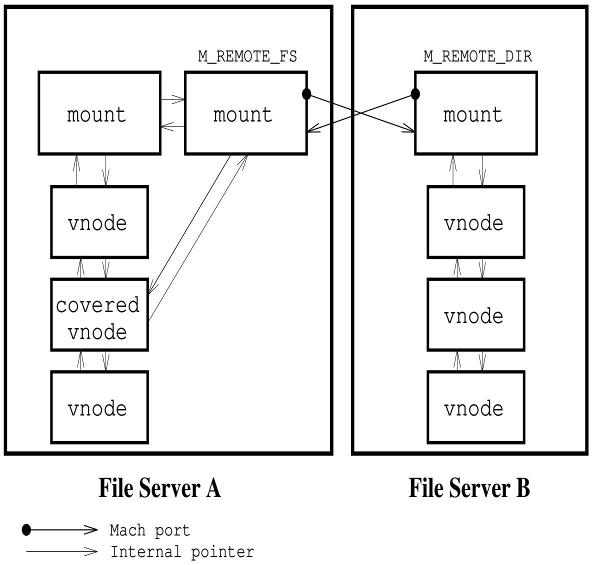

Figure 4: Remote Mount Data Structures

The `mount()` operation is targeted at the file server managing the directory being mounted on. Based on the node information stored in the special file representing the file system being mounted, the mount code determines whether a remote mount needs to be set up. If so, a mount structure is created and marked as `M_REMOTE_FS` (meaning the file system is remote), a port representing it is allocated, and a request containing a send right to this port is sent to the remote file server. Upon receipt, the remote file server creates a mount structure, marking it `M_REMOTE_DIR` (meaning the directory is remote), and stores the send right received as a parameter in the `m_remoteport` field. A port representing the new structure is allocated and a send right to it is returned to the original server, which then completes the operation by storing it in its `m_remoteport` field. (Further information on remote mounts may be found in [Paciorek 92a].)

Pathname translation works exactly as in standard OSF/1 (including pathname caching), until a remote mount point is encountered. At that point, the remainder of the pathname is forwarded to the remote server so that translation may continue. This process is repeated until the pathname is fully translated.

Client Access to the File System

As in standard OSF/1, processes access files via system calls. In a multicomputer environment, it is extremely important to implement these system calls in a manner that minimizes IPC messages and server context switches. OSF/1 AD TNC accomplishes this using the Mach system call redirection mechanism (described in the Background Section) to implement functionality in a per-process *emulator* residing in each process's address space. In particular, the emulator:

- contains the per-process file descriptor table, where each table entry has a port right mapping to the open file structure on a file server. Hence, operations on open files communicate with at most one server.
- contains send rights to the ports representing the root and current working directories for the process. Hence, most (but not all) operations on pathnames (e.g., `open()`, `mkdir()`) communicate with one server.
- provides a location to implement mapped access to open files and accrue the benefits of node-local file data caching (see the next section).

File servers export open file and directory vnode data structures by associating ports with them and providing clients with send rights. Messages received on ports are then converted to function calls on the corresponding data structures.

System calls that perform pathname translation are executed using a *remote system call* model; i.e., messages to file servers instruct the server to resolve the pathname and execute the operation. If, during pathname translation, a remote mount is encountered, the system call is forwarded to the file server servicing the mount, which then continues the translation. This process is repeated until the pathname is fully translated (to a corresponding vnode), at which time the operation is executed.

The emulator determines whether to send an initial system call message to the root or current working directory port depending on whether the pathname is absolute or relative. This effectively acts as a two-entry cache, but can still result in server-to-server messages when remote mount points are crossed during pathname translation. A planned enhancement is to expand this caching mechanism to include a pathname prefix table [Cheriton 89] [Julin 91] [Welch 86].

An alternative to the remote system call model would have been to resolve the pathname (to a vnode) and execute the system call as separate operations (analogous to the VFS interface internal to Unix [Kleiman 86]). However, avoiding extra messages due to pathname translation would have required a pathname caching scheme that was deemed infeasible within the context of the emulator.

The `open()` system call executes as described above, but in addition a send right for the open file structure is returned to the emulator. The emulator stores this right in its file descriptor table so that subsequent system calls for the open file are sent directly to the proper file server.

The file descriptor table is implemented such that a `dup()` system call effectively increments an internal data structure reference without contacting a server. When a process `fork()`'s, the emulator is automatically duplicated (copy-on-write) in the child address space (including the file descriptor table data structures) but send rights to open files must be explicitly inserted into the child process using the Mach kernel interface `mach_port_insert_rights()`. [Loepere 92] Access to a shared file offset between parent and child processes is not a problem because the offset is stored in the common file structure within the file server (except for Unix regular files which are managed as described in the next section).

Closing files is accomplished using Mach's *no-more-senders* notification facility [Draves 90]. File servers request that they be notified (by the kernel) when no more send rights exist to an open file structure. At `close()` time, the emulator simply deallocates the send right it has associated with the open file. When all such rights have been deallocated (meaning no clients have access to the open file structure) the file server will receive a notification and perform tear down as necessary.

A more detailed description of the information presented in this section may be found in [Paciorek 92a] [Paciorek 92b].

Unix File Caching via Mach Memory Objects

The Mach kernel manages physical memory as a cache of memory object contents; access to the actual memory (i.e., backing storage) is via a Mach port (known as a *memory object port*) to which messages can be sent containing data or requesting that it be supplied. This allows memory objects to be implemented by user-state programs such as a file system server or database application.

OSF/1 AD TNC uses Mach memory objects to cache data from Unix regular files. File servers provide backing storage for memory objects using the standard OSF/1 Unix File System (UFS). Unix `read()` and `write()` system calls are converted to mapped accesses to the corresponding memory objects. Cache hits to already-resident data are satisfied without file server interaction. File servers provide *external memory management* by interacting with the Mach kernel via a protocol known as the *External Memory Management Interface* [Young 87] [Young 89]. This protocol is used to fulfill cache misses, clean pages on behalf of `sync()` , flush pages on behalf of `truncate()`, etc.

The Mach 3.0 4.3BSD Single Server and Chorus/MiX V.4 systems [Dean 92] have implemented similar file caching schemes for single node systems. OSF/1 AD TNC operates in a multicomputer environment using Mach's NORMA XMM kernel subsystem (described earlier) to implement cache management across nodes (i.e., distributed shared memory). Accesses to mapped memory objects are performed by the per-process emulator. A file's memory object port is obtained at `open()` time and stored in the emulator's file descriptor table.

Implementing POSIX file I/O semantics [IEEE 90] requires `read()/write()` atomicity and proper update of a file's accessed and modified times. OSF/1 AD TNC implements these via a message-based token protocol between emulators and file servers. A token is associated with each open file and guarantees multi-reader/single-writer access to the file's:

- data, including its length
- seek pointer
- accessed and modified status.

Access to a file's data, and accessed and modified status are synchronized across all processes accessing the file, whereas access to a seek pointer is synchronized only across the processes sharing an open file.

A token (and a file's length and seek pointer) is obtained in a file server's reply to an open request, and is stored in the file descriptor table. When a process executes a `read()` or `write()` system call, its emulator first checks that it still has the token. If so, it proceeds to access the memory object at the offset specified by the seek pointer (or, at the end-of-file if it's an append-mode write). The emulator code must also prohibit reading beyond the end-of-file.

After a successful access, the emulator updates the file's seek pointer, the file's length (if necessary), whether the file was read (accessed) or written (modified), and stores the information in the file descriptor table.

Subsequent I/O's may find the token still held and proceed as above. However, it's also possible the token has been revoked, in which case a message to the file server is required to reacquire a token.

A file server revokes a token by sending a message to an emulator's callback thread, instructing it to release the token by performing a "write-back" of the cached state. A token is revoked either because another process is attempting to acquire the token on behalf of a `read()`, `write()`, or `lseek()`, or because the file server needs the token on behalf of some other operation. For example, `truncate()` requires mutually exclusive access to the file, `sync()` must determine if there are dirty pages to be cleaned, and `stat()` must determine whether the file has been accessed or modified so that the proper times may be reported.

Note that POSIX semantics for updating a file's accessed and modified times only require that they be updated during the last `close()` of the file, or if the file's attributes are read (due to a `stat()` or `fstat()`). Thus, emulators retain booleans indicating whether a file has been accessed or modified, relying on file servers to revoke tokens and update the times as necessary.

If a file with a token is `close()`'d, the emulator sends a message to the file server releasing the token and its associated state.

Tokens are implemented using Mach ports. If a process aborts abnormally, file servers will receive *no-more-senders* notifications for all outstanding tokens, allowing them to perform necessary garbage collection.

The bulk of the code implementing the token mechanism resides in a file server module known as the *mapped files module*. It exports message-based interfaces for emulators as well as internal interfaces for cache management and synchronization. The standard OSF/1 file system code has been modified to invoke these interfaces, with most of the changes isolated to the Unix File System (UFS). Fortunately, these changes are layered in such a way that they can eventually be implemented in a separate file system stacked on top of the UFS.

Remote Device Handling

In standard OSF/1, devices such as tty's and disk partitions are identified via device special files in the file name space. As described previously, a multicomputer environment must have knowledge of the node to which a given device is attached, and hence node numbers are stored in device special files' inodes.

The problem becomes one of efficiency. In particular, if the file server managing a device's special file is also managing the actual device then it becomes a bottleneck because most special files reside in the "/dev" directory. In addition, it's desirable for a file server to be co-located on the same node with the devices it's managing.

OSF/1 AD TNC solves this problem by allowing different file servers to manage a device and the special file representing it. The result is that accesses to an open device communicate only with the file server managing it.

As for `open()`'ing a device, the pathname is resolved to the file server managing the special file, which then forwards an open request to the file server managing the device, passing along a port send right to be used for subsequent communication back to the file server managing the special file. The device is then opened and a port send right to the file structure is returned to the process' emulator which stores it in its file descriptor table. Thus, at completion of open processing the emulator has a direct connection to the file server managing the device, which in turn has a connection to the file server managing the special file.

In order to maintain correct accessed and modified times in a special file's inode, the system relies on POSIX semantics [IEEE 90] which only requires that the times be updated during the last `close()` of the device, or if the device's attributes are read (due to a `stat()` or `fstat()`). Thus, a file server managing an open device retains booleans indicating whether a device has been accessed or modified. This information is propagated back to the file server managing the special file (using the connection established at `open()` time) either via *push* (at last `close()` time) or *pull* (at `stat()` time).

`fstat()` requires that both file servers be contacted (a request message is sent to the file server managing the device, which must contact the file server managing the special file to obtain all the attributes). Avoiding this message would require full caching of a special file's inode in the file server managing the device (as is done in [Batlivala 92]), but the payoff was not deemed worth the extra complexity.

Credentials Service

In a monolithic Unix system, process management and file system functionality are co-resident in the same address space. In OSF/1 AD TNC, the functionality may be partitioned across multiple address spaces, perhaps residing on different nodes in the multicomputer. The credentials service was created to provide a mechanism for bidirectional exchange of information between process managers and file servers. In particular, the credentials service supports the propagation of per-process information from:

- process managers to file servers, including process id, process group id, session id, credentials, file creation mask, and file size limits.
- file servers back to process managers, currently only the file-related resource usage information (blkin, blkout).

The design of the service is based on a master-slave relationship: each process manager has a co-resident master credentials server, while each file server has a co-resident slave credentials server. Instances of each of these are known as the *credentials master* and *credentials cache*, respectively.

Each credentials master maintains associations between its processes and their "keys", known as *credentials ports*. When a new process is created, a credentials port is allocated and provided to the process' emulator. All subsequent messages between the emulator and file servers contain the process' credentials port, thus allowing the file servers to map the credentials port to the relevant process information using the credentials cache. If a process has previously communicated with a particular file server then the process information is readily available, otherwise the credentials cache obtains the information from the credentials master and caches it for future use.

When a process exits or executes the `rusage()` system call, its credentials master collects resource usage information from all relevant credentials caches (i.e., the ones that are caching information for that process). In addition, the credentials caches are updated with changed information (e.g., change of process group id).

Credentials ports are implemented using Mach ports such that credentials masters hold receive rights and emulators and credentials caches hold send rights.

Interruptible File-Related System Calls

In OSF/1 AD TNC, interrupting a system call in progress (e.g., at signal delivery or `exit()` time) is more complex than in standard OSF/1 because process management and file system functionality may be implemented by different servers. This necessitates a mechanism allowing process managers to track down system calls in progress at remote file servers.

This is accomplished by registering file-related system calls in progress in both emulators and servers. When a system call message is about to be sent by an emulator it records the destination port of the message and a unique identifier acting as a transaction id. When a system call is received by a server the identity of the thread executing the operation is stored in a table entry keyed by the destination port and transaction id.

Then, when a process manager decides to interrupt a process's system call in progress (OSF/1 semantics are that only the first thread in the task is interrupted), an abort message is sent to the process's emulator callback thread, providing the identity of the thread whose system call should be aborted. The callback uses this information to determine which message should be interrupted, and then sends an interrupt message to the destination port recorded in the emulator, passing the transaction id as an argument. Upon receipt of this message the server is able to locate the thread executing the operation (if it's still in progress) and cause it to return with EINTR (e.g., by waking it up from an interruptible wait). The transaction id is used to distinguish between multiple messages that may have been sent to the same destination port.

The case of a reply to a system call message "crossing in the mail" with an interrupt message from an emulator is handled via a retry mechanism in process management code that initiates a new interrupt sequence every one second or so until the operation is interrupted (or completes normally).

If a process calls `exit()` (or aborts abnormally), all system calls executing on behalf of the process must be interrupted, and the process's emulator mustn't be relied upon because its integrity is unknown. This is done using the credentials service (described in the previous section) which knows about all file servers that have executed system calls on behalf of a particular process. Messages are sent to the relevant file servers instructing them to abort all system calls executing on behalf of the exiting process, with a reliance on retries (as above) to handle race conditions.

Distributed `select()`

Both `select()` and `poll()` may operate on a set of open files that are serviced by multiple file servers. The emulator handles this distribution by initiating separate select requests on each open file port to determine if the condition has been met. Additionally, if the system call is able to block (due to a specific or indefinite timeout), then each server will be provided with an extra reply port argument so that it may reply to the emulator if and when the condition is met. Upon meeting at least one condition, or timing out, the emulator destroys the reply port and returns from the system call. When the port is destroyed, *dead name notifications* [Draves 90] are generated to all file servers with select operations pending for this particular `select()` system call, allowing them to clean up.

NFS Support

OSF/1 AD TNC allows a multicomputer to provide both NFS client and NFS server services [Sandberg 85]. Given the intimate relationship between NFS and the networking code, it's required that all NFS code be running on the same node (and within the same server) as an instance of the networking code.

Mounting an external NFS server into the OSF/1 AD TNC file name space is similar to mounting local file systems as described in the Distributed File Name Space section. The difference is that the "device" mounted on top of a directory is not a file system device but rather a "NFS device," where the *node number* for that device is a node that is known to be running a server containing both NFS client and networking code. If the directory is managed by a different server, then a remote mount is established in the same way as for local file systems. At this point, all accesses to the mounted NFS server "just work", in that system calls are automatically forwarded across the remote mount to the NFS client code.

NFS server services are provided by running NFS daemons (*nfsd*'s) (nfsd's can run on any node of the multicomputer). An nfsd receives requests from external clients and translates them into requests to the global OSF/1 AD TNC file system. It does this by encoding the node number of a file in the NFS file handle, and forwarding client requests (via Mach IPC) to the file server on that node. A thread in the file server handles the request by executing NFS server code just as if it was an nfsd. After execution, the reply is sent back to the nfsd which then replies to the client.

Socket Enhancements

There are two classes of sockets used in OSF/1 AD TNC: Unix domain sockets and network domain sockets. OSF/1 AD TNC has enhancements to deal with each of these classes:

- *Unix Domain Sockets.* Performance criteria dictate that the data storage for connected pairs of Unix domain sockets, which are used to implement pipes and FIFOs in OSF/1, be kept on the same node as the "primary reader" process. There may be multiple processes writing to a pipe, but typically there is only one process reading from a pipe. Additionally, keeping connected pairs of sockets on the same node reduces changes to the low-level Unix domain protocol code, which assumes all processes communicating through Unix domain sockets share the same server address space.
- *Network Domain Sockets.* There are typically several network-capable nodes in the multicomputer environment. For performance reasons, each of the network-capable nodes executes its own network protocol stacks (this can also be employed to provide some level of fault tolerance).

Virtual Sockets

To accommodate the goals of both true networking protocol families and the Unix domain protocols, a virtual socket layer has been added to OSF/1 AD TNC. A pointer to a virtual socket operation table is stored in each socket when it is created, the operation table chosen being determined by the specified domain, type, and protocol arguments. Socket related system calls have been modified to use macros such as `VSOP_SEND()`, `VSOP_BIND()`, etc., which make indirect calls through the operation table, rather than directly calling `sosend()` or `sobind()`. The virtual socket operation functions themselves hide the multicomputer environment from the socket manipulation code, performing function shipping and coordinating socket activity among the socket-using nodes and network-capable nodes.

Distributed Unix Domain Sockets/Pipes

When a Unix domain socket is initially created, its file server node is the execution node of the process that made the creating system call. In order to have the the socket serviced by a file server on the same node as the primary reader process, the socket must be able to migrate from node to node. This migration is necessary if:

- The primary reader process itself migrates to another node.
- The primary reader process exits, in which case the socket eventually migrates to the node of another reader process. (If there are no other readers, socket migration becomes superfluous since the next
- `ECONNRESET`.)

Socket migration of Unix domain sockets is performed in several phases:

- *Notification* – the migrated process sends a message on those file ports corresponding to Unix domain socket objects.
-
- on the notified file port to complete.
- *Relocation* – port rights and copies of socket data and meta-data are sent to the file server on the new execution node.
- *Arrival* – information sent in the relocation phase is used to reconstruct the socket object on the new node.
- *Clean-up* – data structures and resources on the old node are freed; the file server on the new node takes over responsibility for the socket object.

Although the details of each phase vary depending on the particular type of socket Unix domain socket based object involved, this schema is generally applicable to all of the types – pipe, FIFO, stream, or datagram socket. As the simplest type of socket based object, pipes closely follow the general scenario outlined above, and so for them no further elaboration is presented.

Binding Unix Domain Sockets

In the Unix domain, a socket address corresponds to a name in the file system name space. A socket and path name are presented to the `bind()` system call, which creates a vnode and associates it with the socket. In standard OSF/1, this association is just a pointer to the socket structure in system virtual address space. In OSF/1 AD TNC, the socket and vnode structures may reside on different nodes of the multicomputer, so a Mach port must be used to tie them together. This port is called the socket's *bind port*. When a `connect()` system call issues a `PRU_CONNECT` user request to the Unix domain protocol code, a bind port is obtained using `namei()`. The bind port is then passed to the `VSOP_VSOCKCONNECT()` virtual socket operation to complete the connection in a protocol-specific manner.

Unix Stream Sockets

When establishing a connection with Unix domain stream sockets, the bind port is used as an RPC handle to relocate the connecting socket and its file port to the node where the bound socket resides. The stream socket arrival routine uses pre-existing protocol code to complete the connection, and a new socket file descriptor is returned to an application program blocked in the `accept()` system call. From then on the resulting pair of connected sockets behaves just like a (bidirectional) pipe, and in particular its relocation behavior is the same as a pipe's.

Unix Datagram Sockets

Communication between Unix domain datagram sockets in standard OSF/1 is accomplished by temporarily making a connection between the sockets using pointers in kernel virtual address space. In general these temporary connections are very short-lived, usually only lasting for the transfer of a single datagram. In a NORMA environment, sockets often reside on different nodes. Given the short duration of datagram socket temporary connections, relocating a connecting socket to a bound socket on a remote node for the transfer of so little data would be inefficient.

As with Unix stream sockets, a bind port is used to establish datagram socket connections. However, datagram sockets use the bind port to send actual user data rather than to move the communicating sockets to the same node. Short term, single datagram connections are made by the `sendto()` system call, which discards the bind port send right after the data is sent; longer term connections made with the `connect()` system call keep the send right with the connecting socket until the socket is closed or the connection is explicitly broken by connecting to a different bound socket. Unix datagram sockets follow migrated processes to the new execution node as soon as possible, regardless of whether the process is a reader or a writer.

FIFOs

In Unix systems whose FIFO implementations are based on sockets, an in-core FIFO vnode holds a pointer to a pair of connected Unix domain stream sockets. Opening the FIFO creates a new file port and a file structure with a pointer to this vnode. As with pipes, we want the FIFO storage to follow the primary reader process when it migrates to a new node. This presents special problems for FIFOs. Many file ports may have to be relocated, not just one or two. Because file ports and their file structures reference the FIFO vnode, the vnode itself must also be relocated, but must still be accessible to `namei()` lookups using the same path name. Finally, the virtual socket interface provides little help, since the mechanism of FIFO relocation must naturally center around the vnode layer.

The implemented solution takes advantage of the distributed file system's vnode ports, normally used as the glue that holds remotely mounted file systems on individual nodes together in the global file system name space. Before a FIFO relocates away from the node where its original file system vnode resides, a vnode port is created for it. This *redirect port* is used for forwarding `namei()` lookups. During relocation the receive right for the redirect port moves to the new node, where a "storage node" vnode is allocated. The vnode at the file system node holds a send right for the redirect port, and hooks in `namei()` allow subsequent lookups to be forwarded to the storage node just as lookups that cross inter-node mount points are forwarded.

Although virtual socket operations are not used, the job of moving a FIFO from node to node follows the same pattern, from notification to cleanup. The TNC FIFO implementation uses hook routines similar to virtual socket operations whenever FIFOs are opened or closed, and to receive notification messages and trigger relocation.

Distributed Network Domain Sockets

In a multicomputer, the network protocol processing is distributed among the nodes designated as network servers. A network server may handle one or more protocol stacks for one or more network interfaces. Indeed, network interfaces are not required to be on the same node as network servers. However, in typical configurations, each node with a network interface runs a network server to handle the protocol stacks for that network interface.

The multiple instances of network servers are hidden from the user by the virtual socket layer. When a network server is configured (by configuring an interface), the virtual socket layer notifies a central database, called the *clearinghouse*, with its node number, protocol address (including family), and network server port. The clearinghouse is is responsible for:

- Insuring that two interfaces are not configured with the same address.
- Matching a request from the virtual socket layer for a network server with the best network server available.

Creating Network Sockets

When an application makes a `socket()` system call, a virtual socket is created on the applications node. This socket is referred to as the primary virtual socket.

Binding Network Sockets

When an application makes a `bind()` system call, the clearinghouse is called by the virtual socket layer to determine which network servers to use. If the address is a wild-card address (`INADDR_ANY` or multicast), a list of servers is returned, and the virtual socket is marked as bound to a network server (only wildcard address requests can return more than one network server). A secondary virtual socket is created on each of the network servers returned by the clearing house. If a primary virtual socket and a secondary virtual socket exist on the same node, a single virtual socket is used which functions as both the primary and secondary virtual socket.

Wildcard addresses that request a routing endpoint return only the local node's network server. This is to insure that each network server can run the routing protocols necessary to maintain routing tables. If the address is a single endpoint and the clearinghouse lookup is successful, the virtual socket is marked as bound to a network server. After a socket is bound to a network server, the clearinghouse is not used.

As an optimization, if the bound socket has only one secondary socket associated with it, the virtual socket layer migrates the primary virtual socket and its associated file structure to the node where the secondary virtual socket resides, and the primary and secondary virtual sockets are collapsed into a single virtual socket. This reduces the number of servers which interact when servicing a system call operating on that particular socket.

Connecting Network Sockets

When an application issues the `connect()` system call, and the virtual socket is not bound to a network server by a previous `bind()` system call, the virtual socket layer tries to find the best network server to use for this connection. First the virtual server queries the clearinghouse for the file server of the endpoint the virtual socket layer wishes to connect to; if this fails, the virtual socket layer queries the clearinghouse to get the "nearest" server.

Associating Primary Virtual Sockets with Network Servers

When the virtual socket layer needs to associate a virtual socket with a network server, usually from a `bind()`, `connect()`, or `sendto()` system call, the virtual socket layer queries the clearinghouse for the Mach port of the network server it should use. The clearinghouse contains a set of protocol specific routines based on the family, type and protocol of the socket which match the requested address with a network server. The clearinghouse protocol-dependent match routine is called and returns true if the supplied address matches one or more of the addresses registered in the clearinghouse. From this, a list of network server ports is created.

If no network servers are available to match the request, and the system call does not need an explicit address match (as in the case of `connect()` or `sendto()`), the virtual socket layer calls the clearinghouse with a *getnearestserver* request. The clearinghouse employs an implementation dependent policy to select the network server. A policy function is applied the list of network server / address tuples. Possible policies are to do nothing, return the server for the address with the best (longest) match, return the list in a random order, query each network server to find the best route, etc. The current policy is one which searches the clearinghouse for a connected subnet; if that fails, it sequentially cycles through all the available network servers. An advantage of the current policy is that if two devices reside on the same network, they can be configured on separate network servers. In this configuration, unlike most SMP or uniprocessor implementations, load balancing can be achieved not only on inbound traffic, but on outbound traffic as well, since each connection request will obtain a different server.

Communication between Network Servers

While the user sees the network servers as a single instance of a protocol stack, each network server sees other network servers as if they were connected over a network. This allows protocol stacks to transmit routing information using the protocols' native routing protocols. The *mi* interface driver is used to create this internal network. The *mi* interface is a pseudo interface driver which uses Mach IPC messages to transmit and receive protocol packets among the network servers. The first incarnation of the *mi* driver was a simple point to point driver. However, due to scaling issues, a multipoint driver was developed which uses a central node for replaying broadcast and multicast packets. The clearinghouse provides proxy ARP services for the *mi* interface.

The chief advantage of using the native protocols for propagating routing information, is that as new protocols are developed they can be easily added without modifications to the network servers.

Process Management Enhancements

In OSF/1 AD TNC, process management is distributed across all the nodes of the multicomputer. Typically, processes running on a specific node have their process management system calls serviced on the same node. It is desirable to perform this distribution without major impact on the standard OSF/1 source code. In order to do this, the Virtual Process (or *vproc*) layer was added. By analogy, the *vproc* layer does for processes what *vnode* layer does for the files. The result is location transparency at the system call interface.

A set of new remote processing primitives has been added to enable specialized user processes to exploit multicomputer resources. Through these primitives, processes can fork, exec or migrate onto another node. Furthermore, TNC adds a new signal, `SIGMIGRATE`, that causes a receiving process to move to another node.

In addition, OSF/1 AD TNC provides a load leveling daemon. This daemon makes extensive use of the `SIGMIGRATE` signal to distribute the workload across all the nodes of the multicomputer.

Virtual Processes

Process-related operations are requested through the *vproc* layer [Walker 93]. Only the *vproc* layer is aware of the physical location of – and POSIX-defined relationships between – processes. The *vproc* layer directs each physical process operation to the node where a process is located. The local server is called upon to complete the operation.

For local processes, the code below the vproc layer manipulates the physical process directly. For remote processes, the code below the vproc layer invokes client routines which communicate via RPCs with the process management server on the remote process's execution node to access the physical process on behalf of the client.

Process Identification

A key element of process transparency provided by the vproc architecture is a system-wide process identifier space. Subject to the limitation of 32 bit process identifiers: the high order bits are the node number of the node which created the process; the low order bits are generated uniquely by the creation node. Using this scheme each node can generate PIDs without consulting any other node and yet each PID in the system is unique.

The traditional process 1 (`init`) is treated specially. A single `init` exists for the entire multicomputer system; it is located on the root file server node, but `init` is allocated PID 1 irrespective of which node this may be.

Vproc Structure

The vproc structure is very simple. It is all that is visible of a process to base server code, and the fields in this structure can be relied upon, even if the process is not executing locally nor even known about locally. Details of physical process location, physical process attributes and process relationships are below the vproc abstraction layer and available only through vproc operations. The following is the vproc structure:

```
struct vproc {
    long vp_magic;
    pid_t vp_pid;
    unsigned long vp_ref_cnt;
    struct vproc_ops *vp_ops;
    caddr_t vp_data;
    vp_lock_t vp_ref_cnt_lock;
    struct pid_hash vp_pid_hash;
}
```

- `vp_magic` is a magic number used to ensure that a `struct vproc` is being accessed.
- `vp_pid` is the standard process identifier.
- `vp_ref_cnt` is a local reference count to this data structure. When the count reaches zero, the vproc is freed
- `vp_vops` is a pointer to a table of operations applicable to the vproc.
- `vp_data` is pointer to private data, much like one sees in a `struct vnode`
- `vp_ref_cnt_lock` is a lock for multiprocessor locking of the .T vp_ref_cnt.
- `vp_pid_hash` is used to provide fast access to a vproc for a given PID.

Vproc Operations

Vproc operations hide the distributed implementation of processes and inter-process relationships and are necessary if:

- the operation is to be performed on a process other than the current process; or
- the operation changes or affects inter-process relationships; or
- the operation may change the status of a process in a way which may be of interest to other processes.

In OSF/1 AD TNC there are 18 such operations. One example is VPOP_SIGPROC() which sends a signal to a specified virtual process; this operation is used widely throughout the server and, in particular, in support of the `kill()` system call.

In addition to the operations on individual virtual and physical processes, there are also some operations that operate on the virtual process system as a whole. These operations are known as *virtual process system operations*.

In order to shield virtual process operations from the details of the actual physical process implementation, virtual process operations interact with individual physical processes using a well-defined set of routines, known as *physical process operations*.

Distributed Process Management

A vproc is independent of the execution node of the corresponding physical process and the distribution of related processes (e.g. parent). Those details are hidden within the vproc layer in a private data called `struct pvproc`. On nodes where the process is not executing there could be a vproc and pvproc but there will be no process structure. Figure 5 illustrates the distributed process architecture.

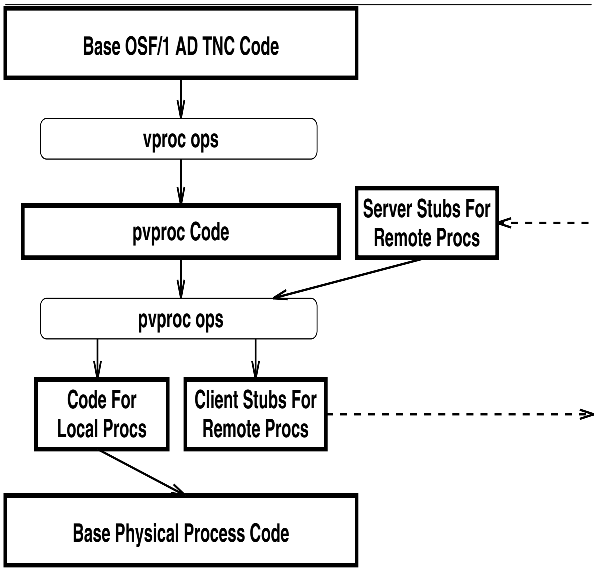

Figure 5: Distributed process architecture

Pvproc Structure

The `pvproc` structure is pointed to by the `vp_data` field in the `struct vproc`. The pvproc structure in turn points to the physical process structure, `struct proc`. Information about process relationships (parent-sibling-child, process group and session membership) resides in the pvproc structure and not the proc structure, which is not referenced or modified directly by the TNC distributed processing code.

Pvproc Operations

Pvproc operations hide the process execution location. Each pvproc operation is executed against a specific process. A pvproc operation may or may not be called on the node where the specified process is executing. However, the operations vector for the pvproc either directs execution to physical process code (*pproc operation*) to act on a locally running process or to a routine called the remote pvproc operations client stub.

The remote pvproc client stub first determines the process's execution node. This may have to be obtained from the process's origin node. The operation is then packaged as a RPC to the execution node. OSF/1 AD TNC associates a Mach port with each vproc, and it is to this vproc port that RPCs are directed.

On the server side of the RPC, a check is made to ensure that the target process is indeed executing locally (it may have moved on). If not, an error is returned and the client code retries by redetermining where the process is executing. If the process is local, the appropriate pvproc operation is invoked, with the return value being sent back to the client.

Private Virtual Process System Operations

This form of operation is directed to be performed upon a specific node rather than upon a specific virtual process. The operations vector for *pvps operations* directs execution to act on the local node or via an RPC to the requested remote node. Such operations are principally internal to the vproc layer; for example, to obtain the current execution node of a particular process from the origin node of that process (determined from its PID).

Remote Processing Primitives

The following additional system calls are provided by OSF/1 AD TNC:

`rfork()`

Taking an additional node parameter over the conventional `fork()` system call, `rfork()` forks a child copy of the caller onto a remote node. As usual, the parent is returned the PID of its new child, but note that this will reflect the number of the node on which the child has been forked.

In all respects, the parent and child behave as if they are co-located: the child inherits its parent's process group id (PGID), session id (SID), text and data segments, open files etc. The parent can wait on its remote child in the usual way.

Remote forking exploits the Mach 3.0 `norma_task_create()` primitive to create the physical child task on the remote node inheriting the memory of the parent. The vproc layer manages internode process relationships by adding a vproc for the new child process in the parent-sibling-child list on the parent node. This child vproc provides a reference to the execution node of the child process. Conversely, a vproc for the parent process on the execution node of the child provides a link back to the execution node of the parent. In a similar manner, process group and session relationship information is maintained for the new child process.

`rforkmulti()`

This form of remote fork is provided to create multiple children at once on a specified set of remote nodes as a more efficient alternative to a series of `rfork()`'s. `rforkmulti()` takes an input array parameter specifying the nodes on which remote forks are to be performed, and returns two arrays indicating the PIDs of the children and status values from each node.

Efficiency is obtained using a binary tree algorithm with the list of nodes being successively divided in half. On each remote node visited, the first element in the list is the node's own node number. Before performing the fork on the visited node, the remaining list is again divided into two parts and `rforkmulti()` is called recursively and asynchronously for each part. Completion of the two asynchronous parts is awaited after the local fork is complete.

`migrate()`

This system call relocates the calling process to the remote node specified by a single parameter. Upon successful return, the caller will be executing on the new node but in all other respects the process is unaltered: it has the same PID, PGID and SID, text and data, open files etc.

Process migration exploits the Mach 3.0 primitive `norma_task_create()` to fashion a new user task on the remote node from the image of the former user task. Process state information about the physical process is transferred to the remote node so that the process is replicated exactly in the process management server on the new node before the process is deleted from the old node.

Process relationship information is also transferred to the new node: If the process is a parent, the list of vprocs representing its child processes is transferred; if a process group leader, the list of vprocs representing the group member processes; and, if a session leader, the list of vprocs representing the process group leaders in the session. Note that a vproc will be deleted from the old node only when no more references to it exist there.

`rexecve()`

Taking an additional node parameter over the conventional `execve()` system call, `rexecve()`

executes a new program image on a remote node. Note that the PID is retained when the process moves to the new node.

A remote exec is performed as a special case of a migration followed by an exec. Process relationship information is handled exactly like a migration but less physical process state information needs to be transferred, since much of this is reset upon executing the new process image.

`kill3()`

This system call is a superset of the usual `kill()` system call. Taking an additional argument, `kill3()` sends a signal and the associated integer argument to a specified process. This semantic is of relevance only to the OSF/1 AD TNC-specific signal `SIGMIGRATE`: the default handling of which is to migrate the receiving process to the node indicated by the associated argument. Hence, `kill3()` may be used to migrate any process regardless of it being aware of the fact that migration is occurring; the load-leveling daemon employs this mechanism to migrate processes away from loaded nodes.

Load Leveling

The function of the load leveler is to balance the load among the multicomputer nodes. In OSF/1 AD TNC, load leveling is accomplished by periodically migrating processes from overloaded nodes to underloaded nodes.

There are two aspects to load leveling:

- Exchanging load information between nodes, in order to determine overloaded and underloaded nodes.
- Migrating processes from overloaded nodes to underloaded nodes.

The OSF/1 AD TNC load leveler comprises daemon processes that run on every node of the system. Each daemon process is responsible for both exchanging load information between nodes and for determining the processes to be migrated from overloaded nodes to underloaded nodes.

Load Information Exchange

The ideal method of maintaining the load information in a multicomputer would be to keep the current load information about all nodes in the multicomputer. In this way, the process migration policy can be very accurate. Unfortunately, maintaining global load information in a very large multicomputer configuration can be very expensive. Hence, a simpler, less expensive algorithm was required. The load information exchange algorithm for OSF/1 AD TNC was derived from the algorithm used by the MOS multicomputer [Barak 85b]. The load leveler daemon process for each node maintains a fixed-size load vector that is exchanged between nodes (the size of the vector is a tunable parameter). This load vector contains load information about an arbitrary (random) subset of nodes in the multicomputer.

Periodically, the load leveler daemon on each node:

- Updates its own load value, putting the load value into the first element of the load vector. The load value is a (tunable) weighted average of the node's 5-second, 30-second, and 1-minute load averages.
- Sends the first half of its load vector to the load leveler daemon process on a random other node.

When the load leveler daemon process receives a portion of a load vector from another node, it discards the latter half of its current load vector and literally "shuffles in" the received portion of the load vector with the first half of its current load vector.

This load information exchange algorithm has been shown to be efficient [Barak 85b], and (with carefully tuned parameters) allows the process migration policy to perform its work accurately.

Process Migration Policy

At periodic intervals, the load leveler daemon on each node calculates the load average of all the load values contained in the load vector. It then uses this average to determine whether the local node is overloaded and whether any other nodes specified in the load vector are underloaded. The interval is a tunable parameter, along with the values used to determine whether nodes are overloaded or underloaded. If the local node is overloaded and at least one other node specified in the local load vector is underloaded, the local load leveler daemon will attempt to migrate processes to underloaded nodes.

The underloaded node that is to receive a specific extra process is probabilistically selected from the list of underloaded nodes (while any underloaded node may be selected, the more underloaded a node is, the higher the probability that it will be selected). The daemon migrates the process by invoking the `kill3()` system call, which sends a `SIGMIGRATE` signal, along with the destination node number, to the process. Processes are selected for migration until the local node is no longer overloaded, or until no underloaded nodes remain.

Booting and Configuration

OSF/1 AD TNC servers configure themselves using *boot config variables* provided to each node when the system boots. These variables are tailored specifically to each node, telling servers running on the node:

- whether to export paging services (so that a file server may provide paging storage for processes and servers on diskless nodes)
- whether to import paging services, and if so from what node
- what node is running the file server managing the root
- what node has the root file system device

attached
- what nodes are network servers
- a list of all nodes in the multicomputer
- a mapping of device node numbers to file server node numbers (so that a file server and the devices it's managing need not be co-located)
- miscellaneous information such as the file name of the emulator image, debug options, etc.

The boot config variables are specified in a *configuration file* maintained by a system administrator. They are loaded into the kernel at boot time (see below) and made available to user-space via the `host_get_boot_info()` kernel interface.

The initial loading of boot config variables into the kernel is inherently machine-dependent. For instance, Intel's Paragon and Hypercube machines use a machine-dependent mechanism for loading the boot config variables as well as kernel, server, and emulator images. Another environment consists of a cluster of 386 clones for debugging (simulating a multicomputer) and uses a special server on the Ethernet to respond to requests from booting kernels for their boot config variables.

Given a node number, locating the corresponding file server relies on a special server, known as the *local node nameservers*. An instance of the local node nameserver runs on each node providing file service, and is registered with a *well-known* name in its local kernel. A file server, in turn, is registered with its local node nameserver. Since a server running on one node is able to look up the local node nameserver running on any other node, the system is thus able to map from a node number to a port for the node's file server. Caching is done to expedite subsequent attempts to locate the same file server.

Related Work

The architecture of an operating system designed to run on a multicomputer system is very similar to the architecture of an operating system designed to run on a networked cluster of systems (generally referred to as a distributed operating system). Indeed, OSF/1 AD TNC was originally developed upon (and still runs upon) a network of 386 clones.

There have been several distributed Unix projects in the past. Several of these projects are surveyed in this section.

The Locus Distributed Operating System [Walker 83] [Popek 85], commercialized by IBM as TCF (Transparent Computing Facility) [Walker 89], was the first commercially available system to provide full single system image in a distributed environment. TCF is a modified version of Unix System V.2, and is only available from IBM. TCF scaled only to a few dozen nodes.

IBM and Locus also have designed a distributed operating system that was based on extensions to DCE [Walker 92]. This system utilized many of the concepts developed in TCF. This system was designed to scaled to hundreds or thousands of nodes, but is not commercially available. The process management enhancements of OSF/1 AD TNC are based on the concepts developed in this system and in TCF.

Sprite [Ousterhout 88] [Douglis 90] is a research system of the University of California at Berkeley designed for a network of homogeneous workstations. Sprite provides a single file system image throughout the workstations in the cluster (using a custom distributed file system) and provides load leveling by use of process migration [Douglis 87].

Amoeba [Tanenbaum 91] [Douglis 90] is a research system at the vrije University of Amsterdam. It is designed for distributed computing and highly parallel applications. It provides only partial Unix compatibility.

MOSIX [Barak 85a] is a research system at the Hebrew University of Jerusalem. It is a Unix-derived distributed operating system that provides load leveling by use of process migration. The OSF/1 AD TNC load leveling algorithms utilize concepts developed in MOSIX [Barak 85b]. MOSIX does not provide a true file location transparency, and does no remote file caching.

Chorus [Armand 89] [Guillemont 91] [Batlivala 92] provides a microkernel and servers that support Unix and distributed processing. However, Chorus does not as yet support single system image across a cluster of nodes, nor does it support load leveling.

Implementation Status

OSF/1 AD TNC is the basis for the Intel Paragon XP/S Supercomputer OSF/1 operating system [Intel 92]. Intel Paragon XP/S Supercomputers systems with more than 512 nodes have been shipped.

Additionally, OSF/1 AD TNC runs on clusters of 386 clones on an Ethernet, simulating a multicomputer.

Future Enhancements

Future enhancements to OSF/1 AD TNC will come in two areas: performance and availability.

Performance

While OSF/1 AD TNC is fully functional, it has not yet undergone rigorous performance analysis and tuning. Detailed analyses must be undertaken to discover and remove performance problems.

Specific areas that may require enhancement are:
- The addition of a prefix name cache to the

emulator, to reduce the number of server-to-server boundary crossings during pathname resolution.

- The addition of file system replication, to allow file server requests for a single file system (such as the root file system) to be load leveled across several nodes.

Availability

The current reconfiguration characteristics of OSF/1 AD TNC are quite simple: nodes cannot enter or leave the configuration without rebooting the whole system. This is entirely a software constraint.

It is desirable to be able to perform reconfiguration without rebooting the system; whether to add a node to the configuration, or to perform a planned removal of a node from the configuration, or to allow for the unplanned removal of a node from the configuration (due to hardware or software problems on the node).

These availability characteristics are especially desirable on larger configurations.

Conclusion

This paper has described the features of OSF/1 AD TNC, a version of OSF/1 suitable for massively parallel multicomputers. OSF/1 AD TNC enables users to take advantage of multicomputer hardware, while retaining full OSF/1 (and thus Unix) semantics. Scalability bottlenecks are avoided by distributing control of the file system, socket protocol stacks, and process management across multiple nodes. Programs may take advantage of the computational resources of multiple nodes via explicit use of new system calls, or may rely on automatic load leveling.

OSF/1 AD TNC is the basis of the operating system for the Intel Paragon XP/S Supercomputer, which has shipped in configurations with more than 512 nodes. Future work to OSF/1 AD TNC will include further performance analysis and tuning, enhancements for greater availability in the face of node failure, and improved reconfiguration characteristics so that nodes may enter and leave the configuration without rebooting the entire system.

Acknowledgements

We thank the folks at Intel's Supercomputer Division for their contributions to OSF/1 AD TNC, including Don Cameron, Charles Johnson, Andrew Pfiffer, Paul Pierce, and Stan Smith. We also thank Joseph Barerra, Jeffrey Heller, Alan Langerman, and Steven Sears for their work improving the Mach kernel.

References

[Armand 89] Armand, F., Gien, M., Hermann, F., Rozier, M. Distributing UNIX Brings it Back to its Original Virtues. In *Proceedings of Workshop on Experiences with Building Distributed (and Multiprocessor) Systems*, October 1989.

[Barak 85a] Barak, A., Litman, A. MOS: A Multicomputer Distributed Operating System, In *Software – Practice and Experience*, Vol. 15(8), August 1985.

[Barak 85b] Barak, A., Shiloh, A. A Distributed Load-balancing Policy for a Multicomputer. In *Software – Practice and Experience*, Vol. 15(9), September 1985.

[Barrera 91] Barrera, J. A Fast Mach Network IPC Implementation. In *Proceedings of the USENIX Mach Symposium*, Nov. 1991.

[Barrera 93] Barrera, J. Copying, Caching, and Sharing in a Distributed Virtual Memory System. Ph.D. Dissertation, Carnegie-Mellon University, to appear 1993.

[Batlivala 92] Batlivala, N., Gleeson, B., Hamrick, J., Lurndal, S., Price, D., Soddy, J., Abrossimov, V. Experience with SVR4 over Chorus. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Architectures*, April 1992.

[Bershad 90] Bershad, B., Anderson, T., Lazowska, E. Lightweight Remote Procedure Call. *ACM Trans. on Computer Systems*, 8, 1, February 1990.

[Black 88] Black, D., Golub, D., Hauth, K., Tevanian, A., Sanzi, R. The Mach Exception Handling Facility. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, May 1988.

[Bricker 91] Bricker. A., Gien, M., et al. Architectural issues in microkern-based operating systems: the CHORUS experience. In *Computer Communications, Special Issue: Platforms for Distributed Applications*, July/August 1991.

[Cheriton 89] Cheriton, D., Mann, T. Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance. *ACM Trans. on Computer Systems*, 7, 2, May 1989).

[Dean 92] Dean, R., Armand, F. Data Movement in Kernelized Systems. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Architectures*, April 1992.

[Douglis 87] Douglis, F., Ousterhout, J. Process Migration in the Sprite Operating System. In *Seventh International Conference on Distributed Computing Systems*, September 1987.

[Douglis 90] Douglis, F., Kaashoek, M. F., Tanenbaum, A. S. A Comparison of Two Distributed Systems: Amoeba and Sprite. In *vrije University report IR-230*, December 1990.

[Draves 90] Draves, R. A Revised IPC Interface. In *Proceedings of the USENIX Mach Symposium*, October 1990.

[Forin 91] Forin, A., Golub, D., and Bershad, B. An I/O System for Mach 3.0. In *Proceedings of the*

Second USENIX Mach Symposium, November 1991.

[Golub 90] Golub, D., Dean, R., Forin, A., Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*.

[Guillemont 91] Guillemont, M., Lipkis, J., Orr, D., Rozier, M. A Second-Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility. In *Proceedings of the Winter 1991 USENIX Conference*.

[IEEE 90] Portable Operating Systems Interface (POSIX) - Part 1: System Application Program Interface. IEEE Std 1003.1, 1990. ISO/IEC 9945-1.

[Intel 92] Intel Paragon XP/S Supercomputer Spec Sheet. Intel Corporation, 1992.

[Julin 91] Julin, D., Chew, J., Stevenson, M., Guedes, P., Neves, P., Roy, P. Generalized Emulation Services for Mach 3.0 - Overview, Experiences, and Current Status. In *Proceedings of the USENIX Mach Symposium*, November 1991.

[Kleiman 86] Kleiman, S. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the Summer 1986 USENIX Conference*.

[Loepere 92] Loepere, K. (editor). *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University.

[OREILLY 91] Guide to OSF/1: A Technical Synopsis. O'Reilly & Associates, Inc., 1991.

[OSF 90] The Design of the OSF/1 Operating System. Open Software Foundation, 1990.

[Ousterhout 88] Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M., Welch, B. The Sprite Network Operating System. In *Computer*, February 1988.

[Paciorek 92a] Paciorek, N., Teller, M., Black, D., Roy, P. Design Specification for an OSF/1 Based Distributed File Server on Mach 3.0. Open Software Foundation Research Institute, 1992.

[Paciorek 92b] Paciorek, N., Teller, M. An Object Oriented, File System Independent, Distributed File Server. In *Proceedings of the USENIX File System Workshop*, May 1992.

[Popek 85] Popek, G., Walker, B. *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[Sandberg 85] Sandberg, R., Goldberg, D., Kleiman, S., Lyon, B. Design and Implementation of the SUN Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*.

[Sansom 86] Sansom, R., Julin, D., Rashid, R. Extending a Capability Based System into a Networked Environment. In *Proceedings of the ACM SIGCOMM 86 Symposium on Communications Architectures and Protocols*, August 1986.

[Tanenbaum 91] Tanenbaum, A. S., Kaashoek, M. F., van Renesse, R., Bal, H. E. The Ameoba distributed operating system – a special report. In *Computer Communications, Special Issue: Platforms for Distributed Applications*, July/August 1991.

[Walker 83] Walker, B., Popek, G., English, B., Kline, C., Thiel, G. The LOCUS Distributed Operating System. In *Proceedings of the Ninth Symposium on Operating Systems Principles*, October, 1983.

[Walker 89] Walker, B., Popek, J. Distributed UNIX Transparency: Goals, Benefits, and the TCF Example. In *Winter 1989 Uniforum Conference Proceedings*.

[Walker 92] Walker, B., Lilienkamp, J., Hopfield, J., Zajcew, R., Thiel, G., Matthews, R., Mott, J., Lawlor, F. Extending DCE to Transparent Processing Clusters. In *Winter 1992 Uniforum Conference Proceedings*.

[Walker 93] Walker, B., Zajcew, R., Thiel, G. Vprocs: A Process Abstraction to Enable Single System Image Distributed Computing. Submitted for publication.

[Welch 86] Welch, B., Ousterhout, J. Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem. *Proceedings of the 6th ICDCS*, May 1986.

[Young 87] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987.

[Young 89] Young, M. Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. Ph.D. Dissertation, Carnegie-Mellon University, November 1989. CMU-CS-89-202.

Author Information

Roman Zajcew is a Senior Scientist at Locus Computing Corporation in San Diego, where he is working in the area of distributed computing. He graduated from the University of Manitoba with Master of Science and Bachelor of Science degrees in Computer Science. Prior to Locus, he worked for Encore Computer Corporation, where he worked on SMP System V. His e-mail address is roman@locus.com.

Paul Roy is a Senior Research Engineer at the OSF Cambridge Research Institute, where he is working on microkernel-based operating systems and new file caching architectures for OSF/1. Prior to OSF, he was in the Operating Systems Group at Apollo Computer, and was a member of the

Distributed Systems Group at Stanford University. Paul received a M.S. degree in Electrical Engineering from Stanford University in 1985 and a B.S. degree in Computer Engineering from the University of Massachusetts in 1983. His e-mail address is roy@osf.org.

David Black is a Senior Research Fellow at the OSF Cambridge Research Institute, where his current work involves a joint research project with Carnegie Mellon University and other collaborators on microkernel based operating system environments. He has also contributed to the design and implementation of portions of the OSF/1 operating system. Dr. Black was one of the key developers of the Mach operating system at CMU, and received the Ph.D. degree in Computer Science in 1990. He also holds an M.S. in Computer Science from CMU, and a B.A. and M.A. in Mathematics and a B.S.E. in Computer Science and Engineering from the University of Pennsylvania. Dr. Black's research interests include microkernel technology, real-time, parallel, and distributed computing. His e-mail address is dlb@osf.og.

Chris Peak has a degree in Mathematics from Cambridge University, England. He is currently a Consulting Member of Technical Staff with Locus Computing Corporation in San Diego. Before moving to the US, he worked for British systems house Logica. He is interested in getting things right. His e-mail address is chrisp@locus.com.

Paulo Guedes received a B.S in Electrical Engineering and a M.S. in Computer Science from the Technical University of Lisbon (IST) and is currently pursuing his Ph.D. in distributed object systems at IST. He was a sabbatical at the OSF Cambridge Research Institute for two and a half years. His interests are in operating systems and object-oriented programming. His e-mail address is pjg@inesc.inesc.pt.

Bradford Kemp is a Scientist at Locus Computing Corporation. He has been active in the development and standardization of of communication protocols at both the international and national levels. At Locus he designed and developed the Virtual Socket Layer. He attended Brown University until 1981 in search of a B.S. in Computer Science. His e-mail address is bhk@locus.com.

John LoVerso is a Senior Research Engineer at the OSF Cambridge Research Institute, where he's working on operating systems for multicomputers. Prior to that he worked on high speed FDDI controllers in the Data Communications Group at Concurrent Computer Corporation, and in the Annex terminal server Development Group at both Xylogics, Inc., and Encore Computer Corporation. He received a B.S. degree in Computer Science from the State University of New York at Buffalo in 1985, and has occasionally worked on a Masters degree ever since. His e-mail address is loverso@osf.org.

Michael Leibensperger is a Senior Member of Technical Staff at the Boston office of Locus Computing Corporation. Before joining Locus he was an unemployed drifter in South Asia; before that he worked in the operating systems and custom products groups at MASSCOMP. He did time at Carnegie-Mellon University. His e-mail address is mjl@locus.com.

Michael Barnett is a Principal Member of Technical Staff at Locus Computing Corporation in San Diego. He graduated from the University of Minnesota and obtained the following three degrees: Bachelor of Electrical Engineering, Bachelor of Physics, Bachelor of Astrophysics. Before joining Locus he worked on system software for Honeywell, Control Data, Lockheed (Space Shuttle Project), and Linkabit/Hughes Network Systems. His e-mail address is mbarnett@locus.com.

Faramarz Rabii has been a Senior Research Engineer at the OSF Cambridge Research Institute since Aug. 1991. Prior to that he was a Senior Software Engineer at Sequent Computer Systems, where he worked on the Dynix Multiprocessor operating system. He also worked at Locus Computing Co. from 1987 to 1990, where he worked on the TCF distributed heterogeneous operating system. He received S.B.'s in Computer Science and Chemistry from MIT in 1981 and 1984, respectively, and an M.S. in Theoretical Chemistry from University of Pennsylvania in 1986. His e-mail address is rabii@osf.org.

Durriya Netterwala has been a Senior Research Engineer at the OSF Cambridge Research Institute since Nov. 1991. Prior to that she was in the Operating Systems Group at Apollo/Hewlett-Packard, where she worked on the Domain/OS Operating System. She received her M.S. in Computer Science from University of Michigan, Ann Arbor in 1987 and an M.S. in Mathematics from the Indian Institute of Technology at Bombay, India. Her e-mail address is durriya@osf.org.