THE V DISTRIBUTED SYSTEM

The V distributed System was developed at Stanford University as part of a research project to explore issues in distributed systems. Aspects of the design suggest important directions for the design of future operating systems and communication systems.

DAVID R. CHERITON

The V distributed system is an operating system designed for a cluster of computer workstations connected by a high-performance network. The system is structured as a relatively small "distributed" kernel, a set of service modules, various run-time libraries and a set of commands, as shown in Figure 1. The kernel is distributed in that a separate copy of the kernel executes on each participating network node yet the separate copies cooperate to provide a single system abstraction of processes in address spaces communicating using a base set of communication primitives. The existence of multiple machines and network interconnection is largely transparent at the process level. The service modules implement value-added services using the basic access to hardware resources provided by the kernel. For instance, the V file server implements a UNIX-like file system using the raw disk access supported by the kernel. The various run-time libraries implement conventional language or application-to-operating system interfaces such as Pascal I/O and *C stdio* [21]. Most V applications and commands are written in terms of these conventional interfaces and are oblivious to the distributed nature of the underlying system. In fact, many programs originated in non-distributed systems and were ported with little or no modification—the original source was simply linked against the V run-time libraries.

© 1988 ACM 001-0782/88/0300-0314 $1.50

The development of V was motivated by the growing availability and functionality of relatively low-cost high-performance computer workstations and local networks. Our basic hypothesis was that an operating system could be developed that managed a *cluster* of these workstations and server machines, providing the resource and information sharing facilities of a conventional single-machine system but running on this new, more powerful and more economical hardware base. This hypothesis contrasts with the conventional single mainframe approach to supporting a user community. It also contrasts with the personal computer approach in which the focus is on individual use; the sharing of information and hardware resources between computers may be difficult, if not impossible. The mainframe solution is less extensible, less reliable and less cost effective than appears possible with good use of a workstation cluster. However, the conventional personal computer approach fragments the hardware and software base, wastes hardware resources and makes system management difficult. As an extreme example, an engineering firm might require a simulation package be available to each of its personnel. Each of its personal computers would require the disk space, memory capability and processing power to run the simulation (as well as possibly the license to do so). Yet, the utilization of hardware and software would be much lower than for a conventional timesharing solution, possibly resulting in a higher cost. Moreover, the personal computer solution would be slower for the cases in which

the full power of the mainframe would have been available, such as running the simulation at night.

A first tenet in our design philosophy is that high-performance communication is the most critical facility for distributed systems. By high performance, we mean providing fast exchange of significant amounts of data matching in essence the requirements of conventional file access. Slow communication facilities lead to poor performance and a proliferation of elaborate techniques for dealing with these limited facilities, analogous to the effect of slow and expensive memory on operating systems technology in the 1960s and 1970s. Fast communication allows the system to access state, such as files, without concern for location, thereby making true network transparency feasible. This is analogous to the liberating affect that low-cost memory has had on operating systems and applications since the late 1970s. Not only are the resulting systems faster, they are also simpler because there is no need to highly optimize the use of communication as a scarce resource.

A second tenet of the design philosophy is that the protocols, not the software, define the system. In particular, any network node that "speaks" the system protocols (or a sensible subset) can participate, independent of its internal software architecture. Thus, the challenge is to design the protocols that lead to a system of performance, functionality, reliability and even security required for the system goal. Given the protocols, their implementation is essentially a software engineering and programming problem. This uniform protocol approach is a central theme in the international standards effort for so-called open systems interconnection but is less recognized in the distributed operating

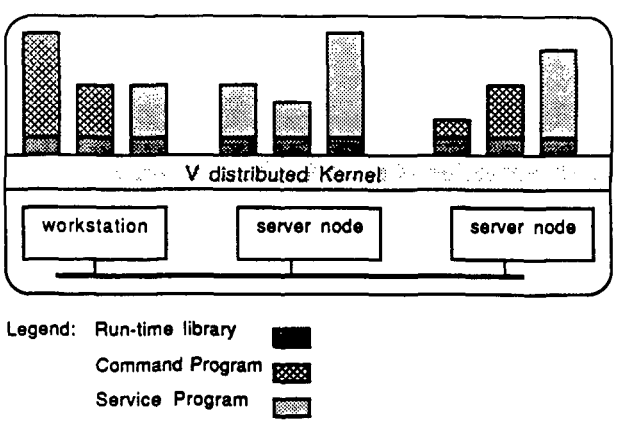

FIGURE 1. The V Distributed Operating System

systems research community. The uniform protocol approach in combination with our belief in the importance of performance requires protocols that are both fast and general purpose; the development of such protocols has been central to our work.

A final major tenet is that a relatively small operating system kernel can implement the basic protocols and services, providing a simple network-transparent process, address space and communication model. The rest of the system can then be implemented at the process level in a machine and network independent fashion. Our goal is for the kernel to provide a *software backplane* analogous to what a good hardware backplane provides for hardware systems. That is, it provides slots into which one can plug modules, power to run those modules, and communication facilities by which these modules can interact, accommodating a wide range of configurations. The design of the system bus and backplane determines many of the major system design decisions and is thus largely responsible for the technical success or failure of the system. With system bus and backplane design, the maximum number and size of the slots, the nature and totality of the power, and the flexibility and performance of the communication facility determine the possible hardware systems that can be build on this chassis. A board designer is limited by these system bus attributes. (On the positive side, a board designer need only design to interface to the backplane in order to have his board interface to the rest of the system, at least at the hardware level.) Our research goal was to understand how to provide a similar base for a distributed operating system running a cluster of workstations, recognizing the performance and functionality requirements of a range of configurations and applications, and the reliability, security and maintainability benefits to keeping the kernel as small as possible.

In the course of this work, two additional ideas significantly affected the design. First, the handling of shared state was recognized as the primary challenge of distributed systems. Shared memory is the most natural model for handling shared state. Shared memory can be implemented across multiple machines i.e., by caching referenced data as virtual memory pages and implementing a consistency protocol between the page frame caches on different machines. The major disadvantage is the cost of consistency operations when contention arises. Thus, we have been investigating efficient mechanisms for implementing consistency between network nodes, software structuring techniques that reduce contention and non-standard forms of consistency that are less expensive than conventional consistency, so-called *problem-oriented shared memory*.

Second, we recognized that modern systems often deal with groups of entities, the same as they deal with individual entities. Examples include the group of users working on a project and the group of processes attached to one terminal. Group support is accentuated further in distributed systems where the sets of file servers, network nodes, printers and other resources constitute additional groups, replacing the single instances of these resources in conventional systems. Applied to the communication domain, interest in group communication has led to support in V for multicast[1] and the development of various group communication protocols.

[1] *Multicast* is defined as sending to a specific subset of the hosts or processes as opposed to *broadcast* which is sending to all hosts or processes on the network.

It was not a goal of the research to develop new programming models or structuring methods for this environment. Conventional programming models are supported; only the underlying implementation is different. Thus, the V operating system appears to the application as a set of procedural interfaces that provide access to the system services. Each system-provided service procedure is part of one of the V run-time libraries. On invocation, the procedure performs the operation directly within the address space if possible. Otherwise, it uses the kernel-provided interprocess communication (IPC) to communicate with V service modules to implement the operation. The implementation of **get-byte** operation, a performance-critical operation in many application environments, illustrates this structure.

The V **get-byte** operation first checks the corresponding I/O buffer in the invoking process's address space to see if there is another byte available in the buffer. If so, it returns the byte immediately. (In fact, this routine is compiled in-line so the cost of the operation in the minimal case is a few microseconds.) If not, it sends a read request to the I/O service module in the file server associated with the open file, as shown in Figure 2. In general, the performance of the I/O operations depends on the kernel providing efficient communication between the application and the various I/O service modules (such as the file servers). It also depends on the interfaces implemented by these service modules and the I/O run-time procedures operating efficiently on top of the communication facility. For example, the buffered implementation of **get-byte** means that the kernel is not invoked on most **get-byte** calls and a large data transfer is made when the kernel IPC is invoked, reducing the overhead per byte transferred. Finally, the various I/O modules must provide a common interface so that the I/O operation can access any one of these modules without special code for each different type thereby handling the wide variety of files and devices available in modern systems. Finally, operations that use character-string names, such as the file open operation, require a way to locate the file based on the name plus an efficient identifier for the client and the server to use once the file is opened for read and write operations.

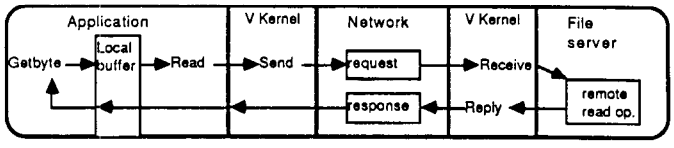

FIGURE 2. Get-byte Invoking a Remote Read Operation

This article describes how V supports efficient file access and other operating system services in terms of the V interprocess communication, I/O, naming and memory management.

THE V KERNEL AS A SOFTWARE BACKPLANE

The V kernel provides a network-transparent abstraction of address spaces, lightweight processes and interprocess communication. These facilities are analogous to those provided by a hardware backplane. The address space corresponds to a backplane slot; a program can be plugged into an address space independent of what is running in other address spaces just as a circuit board can be plugged in. The lightweight process corresponds to the electrical power delivered by the backplane; it is some portion of the power available in the system.[2] Finally, interprocess communication corresponds to the data and control lines provided on the backplane bus, allowing slots to communicate. A good hardware backplane provides the slots, power and communication with the best possible performance, reliability and security for the money, and nothing else.

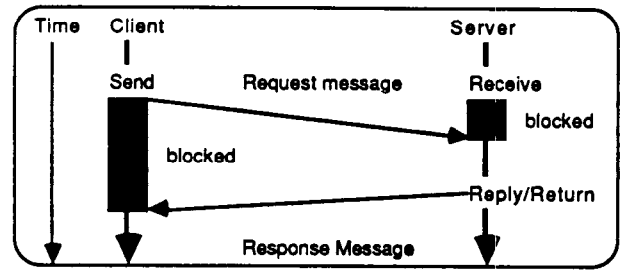

FIGURE 3. Basic Interaction using V IPC

A similar minimalist philosophy was used in the design of the V kernel.

The basic notion of an operating system kernel that only provides an interconnection mechanism for connecting applications to service modules, and avoids implementing the services directly is quite old. For instance, Brinch-Hansen [5] developed such a system, the RC 4000, in the late 1960s. This system, one of the first academically reported message-based systems, was characterized by an elegant design and problematic performance. Other systems followed, including Thoth [9, 14], DEMOS [3] and Accent [28], but performance remained a key concern. A major focus of our research with the V kernel has been exploring ways to achieve good interprocess communication performance.

Interprocess Communication

The kernel interprocess communication facility was designed to provide a fast transport-level service for remote procedure calls, as characterized by file read and write operations. A client **Send** operation sends a request to a server and waits for, and returns, the response. The request corresponds to a (remote) call frame and the response corresponds to the return results. The timing of this operation with respect to the receiving process is shown in Figure 3. The server may

[2] We use *process* in its usual sense as a locus of control that can logically execute in parallel with other processes. *Lightweight* means that each process does not carry the weight of a separate address space. That is, there can be multiple processes per address space, each such process sharing the same address space descriptor. The terms *thread* and *task* have been used as well.

execute as a separate dedicated server process, receiving and acting on the request following the message model of communication. That is, the receiver executes a *Receive* kernel operation to receive the next request message, invokes a procedure to handle the request and then sends a response. Alternatively, the server may effectively execute as a procedure invocation executing on behalf of the requesting process, following the remote procedure call model. In the message model, the request is queued for processing should the addressed process be busy when the request is received. The client process is oblivious to which model is used by the server because the client process blocks waiting for the response in both cases. The message model appears preferable when the serialization of request handling is required. The procedure invocation model is preferred when there are significant performance benefits to concurrent request handling.

The interconnection provided by the V kernel is illustrated in Figure 4. A **Send** to a service module is trapped into the local IPC module if the service module is local. Otherwise, it is processed by the network IPC module, using the VMTP transport protocol [12] to communicate with the remote kernel and remote service module.

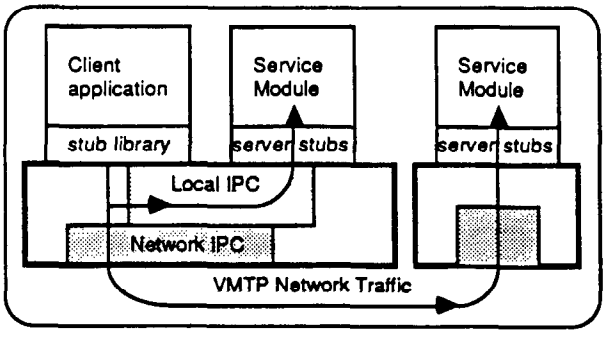

FIGURE 4. Local and Remote Interconnection using V IPC

Fast interprocess communication in V is achieved by: using relatively simple and basic interprocess communication primitives; by using a transport protocol that is carefully designed to support these primitives; by optimizing for the performance-critical common cases; and by internally structuring the kernel for efficient communication. Here we will focus on specific, illustrative techniques we have used to achieve good performance.

One example of the simplicity of the IPC is the request-response behavior of the operations. The client actions of sending a request and receiving a response are combined into the single **Send** primitive. This combination results in one kernel operation for the common case of a remote procedure call, reducing the rescheduling overhead and simplifying the buffering (because the request data can be left in the client's buffer and the response data can be delivered directly into this buffer). It also simplifies the transport-level protocol because both error handling and flow control exploit the response to acknowledge a request and authorize a new request.

Another example is the support for fixed-sized messages (of 32 bytes) with an optional *data segment* of up to 16 kilobytes. As recognized by Almes [1] in his remote procedure call implementation using V, the short message is analogous to the general-purpose registers of a processor for local procedure calls: they introduce some extra complexity to handle well but the resulting performance benefits justify the effort.[3] The handling of the fixed-length messages is optimized at the kernel interface, kernel buffering and network packet transmission and reception. Given that more than 50 percent of our message traffic fits into these short messages [19], this optimization seems appropriate.

Network IPC performance benefits from the use of the VMTP transport protocol, which is optimized for request-response behavior. In particular, there is no explicit connection setup or teardown. In the common case, a message transaction consists of a request packet and a response packet, the response acknowledging the request. Communication state for a client is established upon receiving a request from that client. It is updated on each subsequent request, providing for duplicate suppression as well as caching of information about the client, including authentication information. VMTP also includes the short fixed-size message in the VMTP header, aiding the efficiency of handling small messages. In addition, VMTP supports multicast, datagrams, forwarding, streaming, security and priority. Although VMTP was designed to support efficient V network IPC, we believe it is largely independent of V and suitable for more general use. For instance, we have a UNIX kernel implementation of VMTP that exhibits an 8 millisecond return trip time and 1.9 Mbps data rate between two Microvax II UNIX machines sharing an Ethernet. We are attempting to export our experience with efficient request-response protocols by proposing VMTP as a candidate for a standard transport protocol in the context of the Department of Defense's Internet.

Finally, we have structured the kernel to minimize the cost of the communication operations. For example, every process descriptor contains a template VMTP header with some of the fields initialized at process creation. Using this header, the overhead of preparing a packet as part of a **Send** operation is significantly reduced. In particular, there is no need to allocate a descriptor or buffer for queuing on the network output queue. The fixed-size message is transferred from application to processor registers to the appropriate portion of the process descriptor which is then queued directly for network transmission, as illustrated in Figure 5. Consequently, the elapsed time for transmission of a datagram request is less than 0.8 milliseconds on a SUN-3/75 workstation. Reception of the response is essentially the reverse of these actions.

V IPC performance is given in Table I by the elapsed

[3] Interestingly, processor registers can be used to great benefit to implement efficient IPC [5]. In fact, the 32-byte message is contained in 8 of the general-purpose registers on kernel trap in both the Vax and MC 68000 implementations of V.

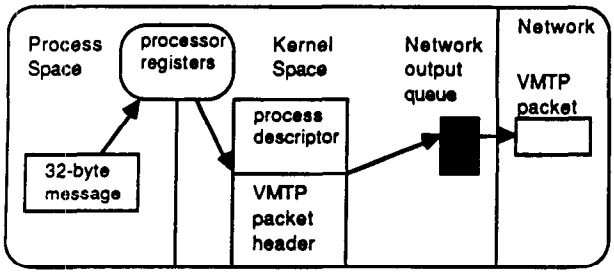

FIGURE 5. Transmission of a 32-byte Request

time for the **Send** operation and the corresponding data transfer rate with various amounts of segment data. (These times were measured with two SUN 3/75's connected by 10 Mbit Ethernet.) The first row gives the time for a basic message transaction exchanging the 32-byte short message and a null segment process to process. The remaining rows reflect the effect of increasing data segment size in a message transaction, both in increased elapsed time and increased effective data transfer rate. In each case, the second column gives the elapsed time to send a 32-byte request and receive a 32-byte response with a segment of data of the size specified in column 1. (These measurements were made with the server idle and insignificant processing time between the time the server receives the request and returns the response.) Table I only lists the times for different sizes of responses, as would occur for file reads, a performance-critical operation. Multi-packet requests with a minimal response, such as arise with file writing, are almost identical in cost.

TABLE I. V Network IPC Elapsed Time (in milliseconds)

Operation (data in Kbytes)	Time (milliseconds)	Data Rate (Mbits/sec.)
0	2.54	0.10
1	3.93	2.08
4	11.2	2.92
8	17.8	3.68
12	23.0	4.27
16	30.0	4.37

Local IPC performance is considerably faster. For instance, a 32-byte request-response to a local server process is 0.480 milliseconds on a SUN-3/75, and a request with an 8 kilobyte response is 2.7 milliseconds versus 17.8 milliseconds for the remote case. However, our primary focus has been on the performance of network interprocess communication. The local IPC performance is regarded as an incidental benefit when modules happened to be co-resident on the same host.

There are additional optimizations which would improve the network interprocess communication further. However, major improvements appear to require significant advances in network interface design; this is one focus of our current research. Moreover, with the level of performance of interprocess communication we have achieved, the system performance appears more dependent on other factors, such as the effectiveness of local and remote file caching. For example, with only a 15.1 millisecond difference between accessing a 8 kilobyte block locally versus remotely, it is faster to access a copy of the block at a remote server that has the data in its RAM cache than to read it from a local disk.

Process Groups and Multicast Communication

Groups of processes arise in a number of settings in V, exploiting the provision of multiple processes per user, per service and per program. Examples include the group of file servers, the group of processes executing a job, and the group of processes executing a single parallel program. V supports the notion of a process group as a set of processes identified by a group identifier (chosen from the same space as process identifiers). A group can have any number of members scattered across any number of hosts. A process can belong to multiple groups.

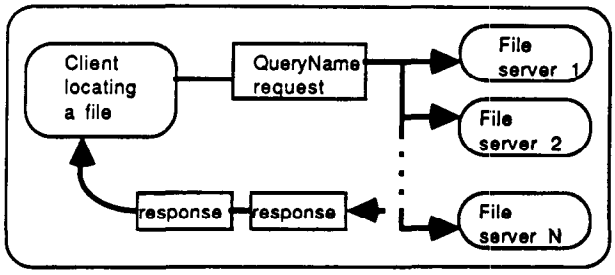

FIGURE 6. Multicast Communication with Multiple Responses

The kernel supports a variety of operations on process groups including the ability to send to a group of processes and receive multiple responses to the request. This multicast communication behavior is illustrated in Figure 6. Besides providing multi-destination delivery, this facility provides *logical addressing* using the extra level of indirection in the name mapping introduced by the process group identification. For example, there is a statically assigned, well-known process group identifier for the group of file servers. This well-known identifier can be used to send a message to a particular file server, using additional discriminating information in the message.

The multicast facility in V is used in a number of ways. Multicast is used to map character string names in the naming protocol, as described in the section on Naming and illustrated in Figure 6. Multicast is used to transmit clock synchronization information in the V time servers. Multicast is used to request as well as distribute load information as part of the distributed scheduling mechanism. Multicast is also used as part of the V atomic transaction protocol and is used in the replicated file update protocol. The level of multicast traffic has been measured in V at slightly less than 1 percent [19]. Each new use of multicast seems to generate significantly more unicast traffic so we do not

expect the relative amount of multicast to increase appreciably in the future. However, like a fire extinguisher, the value of multicast is not entirely represented by frequency use. It has been our experience that when it is used, multicast is extremely useful.

A group **Send** optionally includes a qualifier in the message indicating that the message should only be delivered to those members of the destination group that are co-resident with the process specified in the qualifier. In its most common (and originally motivating) use, a client uses this facility to address a message to the manager of P, where P is a process specified in the qualifier, knowing only the identifier for the group of such managers. For example, process management is handled in a distributed fashion by the group of process managers, one per host. Thus, a suspend operation on a process P is sent to the group of process managers with a co-resident qualifier specifying process P, thereby delivering the request to only the manager in charge of that process, not the entire group. In this fashion, the client is able to address the right manager knowing only the (well-known) group identifier for this group and the process identifier of the process on which it wishes to act. The kernel simply routes the request to the host address for P. One can also specify a process group identifier as the co-resident qualifier, in which case the message is delivered to all members of the destination group that are co-resident with group specified by the co-resident qualifier.

The process group mechanism and multicast communication are used to implement distributed and replicated implementation of services. The kernel-resident servers that manage processes, memory, communication and devices are good examples of such distributed services.

Kernel Servers

The kernel provides time, process, memory, communication and device management in addition to the basic communication facilities. Each of these functions is implemented by a separate kernel module that is replicated in each host, handling the local processes, address spaces and devices, respectively. Each module is registered with the interprocess communication facility and invoked from the process level using the standard IPC facilities, the same as if the module executed outside the kernel as a process, as illustrated in Figure 7. As illustrative examples: a new process is created by sending the request to the kernel process server; a block is read from the disk by sending a request to the kernel device server; a process is added to a process group by requesting this action from the communication server; and a new address space is created by sending to the kernel memory server.

Replicating these modules in each instantiation of the kernel and interfacing to these modules through the standard IPC mechanism has several significant advantages. First, operations on local objects, the common case, are performed fast because the operation is handled entirely by the local server. Also, the implementation of each module is simplified because each instance of the server module only manages local objects, not remote objects. Second, a client can access the kernel servers the same as the other servers (using the same IPC-based network-transparent access), allowing the use of remote procedure call mechanisms and run-time routines that support the high-level protocols. For example, the device server implements the same I/O protocol as other process-level servers and can be accessed using the same I/O run-time support. Use of the IPC interface also minimizes the additional kernel mechanism for accessing remote kernel servers. Third, the use of the IPC primitives to access these servers avoids adding additional kernel traps beyond that required by the IPC primitives. Besides avoiding a proliferation of "system calls", this design simplifies the job of imposing and verifying integrity and security requirements for the kernel. Fourth, this design separates the IPC from other kernel services so that the IPC mechanism, the performance-critical portion of the system, can be tuned independently of these other less performance-critical services. Finally, the invocation mechanism is general in that additional kernel server modules can be added, as might be required in high-performance real-time control systems.

Some portion of each of these services must be included in the kernel to guarantee the integrity of the extended machine implemented by the kernel, and because interrupts must be handled (at least to some degree) by the kernel on most architectures. The V kernel provides a definition of each facility based on maximizing performance, minimizing complexity in the kernel and maximizing the reliability and security attributes of the system.

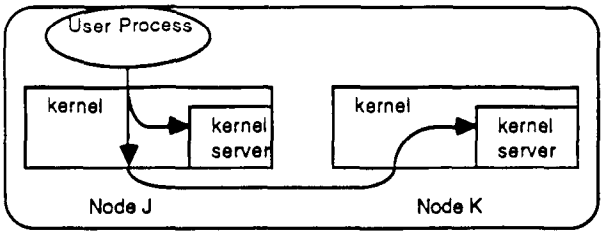

FIGURE 7. Invocation of a Kernel Server

Time

The kernel time service maintains the current time of day (in Greenwich Mean Time—GMT) and allows a process to get the time, set the time and delay for a specified period of time. There is also an operation to wake up a process that is delaying. The synchronization of time service with other nodes is implemented outside the kernel by a process that uses V IPC to coordinate with its counterparts on other machines. The kernel implementation of the time service provides the accuracy of reading the time and delaying required for real-time systems yet does not include the complexity of a kernel-level time synchronization protocol and mechanism.

Process Management

The kernel process server implements operations to create, destroy, query, modify and migrate processes. The primary sources of complication in process management for conventional operating systems are process initiation, process termination, process scheduling and exception handling. The V kernel minimizes the kernel process management mechanism as follows:

First, process initiation is separated from address space creation and initialization (which is discussed in the next section), making the creation of a new process simply a matter of allocating and initializing a new process descriptor.

Second, process termination is simplified because there are few resources at the kernel level to reclaim; most operating system resources, such as open files, are managed at the process level by various server modules. Moreover, the kernel makes no effort to inform servers when a process terminates, further simplifying termination. Each server is responsible for associating each resource it allocates with a client process and checking periodically whether the client exists, reclaiming the resource if not. For example, the file server has a "garbage collector" process that closes files that are associated with deceased processes. Standard V run-time routines executed by the client on normal exit inform the servers to release resources, minimizing the accumulation rate of these "dangling" resources in servers in the common case. In our experience, the garbage collection code in the servers is not significant and the garbage collection overhead is minimal.

Third, scheduling is simplified by the kernel providing only simple priority-based scheduling. A second level of scheduling is performed outside the kernel by a dedicated scheduler process that manipulates priorities to effectively implement timeslicing among interactive and background processes. A number of high priority levels are reserved for real-time processes and operate independent of the scheduler. The priority-based scheduling in the kernel provides simple, efficient low-level dispatching yet is an adequate basis for the higher-level scheduling. Besides simplifying the kernel code, implementing the higher-level scheduling outside the kernel makes the full kernel facilities, including the interprocess communication, available to the process-level scheduler. For example, the process-level scheduler uses multicast communication with the group of such scheduler processes to implement distributed scheduling of programs within the workstation cluster.

The kernel scheduling policy requires the kernel ensure that all K processors are always running the K highest priority processes at any given time. This policy is implemented exactly in a uniprocessor system. However, with multiprocessors, the policy appears to incur excessive overhead. In particular, with a strict implementation of this policy, it appears necessary to check the priority of the process being executed by each processor at the point that a process is made eligible for execution. In our multiprocessor implementation, a process is associated with a processor and its *ready queue*. The low-level dispatching deals only with the priority of processes associated with that processor and its ready queue. A periodically invoked kernel procedure balances the processing load across the processors by migrating processes between processors, which involves simply changing their associated ready queue. Further experience with our multiprocessor implementation is required to determine the adequacy of this approach.

Finally, to avoid the full complexity of exception handling in the kernel, the process management module simply causes the exception-incurring process to send a message describing its problem to the *exception server*, a server provided at the process level, as illustrated in Figure 8. The exception server then takes over, using the facilities of the kernel and other higher level servers to deal with the process. For example, the standard behavior in V is for the exception server to invoke an interactive debugger. With this design, a powerful, flexible and network transparent exception handling mechanism can be implemented at the process level with little kernel complexity.

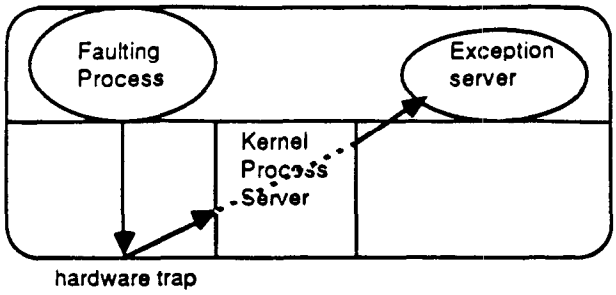

FIGURE 8. Exception Handling

Process migration was retrofitted into V as part of Theimer's Ph.D. thesis project [33]. Modifications to the kernel were relatively modest. Support was provided for extracting kernel-level process information from the originating host and initializing processes and address spaces with the same information in a new host. Also, the ability to freeze and unfreeze processes was added to control the modifications to the address space during migration. The ability to suspend a process in execution was already available by setting the process to a special low priority. The process migration work did, however, point out a number of problems with the kernel design, as described in the section on Kernel Design Mistakes.

It appears that removing any kernel-level process management facilities from the V kernel would result in significant loss of performance, function and integrity. For example, a run-time or user-level implementation of lightweight processes, in place of the kernel implementation, would preclude real parallel execution of these processes on a multiprocessor machine (because the kernel processor scheduling would not know about these processes) as well as introduce the inefficiencies of two-level dispatching and data transfer. Moreover, the V kernel implementation of lightweight processes is quite simple, essentially following the same approach as used in Thoth [9]. The space cost of a process is reduced by concentrating all per-address

space information in a separate address space descriptor, with only per-process information plus a pointer to the address space descriptor located in the process descriptor, as suggested in Figure 9. In this figure, three

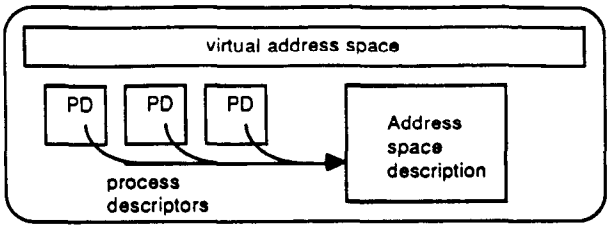

FIGURE 9. Process and Address Space Descriptors

processes are contained in one virtual address space. The space cost is further reduced because of the simplicity of the interprocess communication facilities and other state required in the process descriptor. Also, the kernel is structured so that there is a kernel stack per processor, not per process. The simplicity of the kernel operations means that a process does not need to maintain state on a kernel stack when it blocks as part of kernel operation.

Currently, the V process descriptor is 260 bytes, an acceptable cost given the low cost of memory. However, the current size is larger than strictly necessary because of on-going experimentation with various extended facilities that use extra fields from time to time. The time to create and then destroy a process in V is 4.6 milliseconds on a Microvax II. This time includes the time to allocate a fixed-size stack in the (existing) address space, perform the kernel initialization of the process descriptor and then delete the process descriptor and deallocate the stack.

Memory Management

The kernel must implement some level of memory management to protect its integrity and that of processes from accidental or malicious corruption or unauthorized access, given that encapsulation in virtual address spaces is the primary form of protection used by V, and supported by the hardware for which V is intended. Also, page faults are signaled initially to the kernel.

In the V kernel memory management system, recently extended to support demand paging [11], an address space consists entirely of ranges of addresses, called *regions*, bound to some portion of an open file (or UIO object in the parlance). A reference to a memory cell of a region is semantically a reference to the corresponding data in the open file bound to this region. The kernel serves solely as a binding, caching and consistency mechanism for regions and open files. A page fault is simply a reference to a portion of a region that is not bound at the hardware level to a cached copy of that portion of the bound object. On a page fault, the kernel maps from the virtual address to a block in the bound UIO or open file, and then either locates that block in the kernel page frame cache or else causes the faulting process to send a read request requesting the data block to the server implementing the open file. Physical memory is managed as a cache of pages from these open files. This behavior is illustrated in Figure 10. Consistency is an issue because the block may be stored in multiple page frame caches simultaneously. A simple ownership protocol is used in conjunction with a lock manager at the backing server to implement consistency.

Using this virtual memory system, creation and initialization of address spaces for program execution consists of allocating an address space descriptor and then binding the program file into this address space. The actual transfer of the program file pages and mapping into the address space is handled on demand as the process references portions of the new address space. Thus, there is no special mechanism in the kernel for program loading. In addition, the kernel memory server supports file-like read/write access to address spaces using the system-standard UIO interface, allowing the use of the normal I/O run-time procedures to read and write the address space. This access is used by the debugger to display and modify the debuggee's address space. The V IPC access allows the debugger to run remotely relative to the debuggee with no special provision in the debugger. In the non-demand paged configuration of the kernel, this facility is also used by the migration program to copy the address space to be migrated.

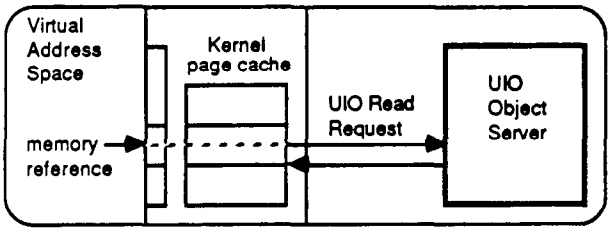

FIGURE 10. Page Fault Handling

An efficient file caching mechanism is provided using the virtual memory caching mechanism in conjunction with a process-level cache directory server. The process-level server maps file open requests onto locally cached open files. Client read and write operations on a cached file use the standard UIO interface to the page cache data implemented by the kernel memory server and are satisfied from the page frame cache when the data is in the cache, otherwise reading the data into the page cache from the file server implementing the real file. Using this mechanism, a one kilobyte read operation satisfied by data in the local page frame cache takes 2.3 milliseconds on a Microvax II, as compared to 8.7 milliseconds to read the page from the backing file server (also a Microvax II), assuming the file server and network are not loaded. The virtual memory mechanism has added 3,300 lines of code and 13.5 kilobytes to the size of the V kernel, out of a previous size of 13,000 lines and 86 kilobytes.

The cost in space and complexity is outweighed by the performance and functionality provided by the kernel-based memory management. In particular, the system is able to make efficient use of large RAM configurations, particularly for diskless workstations, because there is a single cache, namely the page frame cache, for both file and program pages. Thus, the overheads of copying between caches, duplicating data between caches and allocating physical memory between caches are eliminated. Also, processes are able to access the cached data using either the mapped I/O or file read/write paradigms with the efficiency of a direct kernel access path to the data. This design effectively provides a degenerate kernel-based file system that only implements the performance-critical file operations, namely file reading and writing. Directory management, disk space allocation, access control and other conventional and complicated file system functions are placed at the process level, thereby minimizing kernel complexity for this facility at no significant cost in performance.

Device Management

The device server implements access to devices supported by the kernel, including disk, network interface, mouse, frame buffer, keyboard, serial line and tape. The device server module itself is device- (and machine-) independent code that interfaces between the process-level client and the driver modules for the individual devices. The device server implements the UIO interface described in the I/O section at the client interface (on top of the standard V IPC primitives), allowing client processes to use the standard I/O runtime support for device I/O.

The V kernel device support is designed to provide efficient, reliable, machine-independent and secure device access while minimizing the amount of kernel device support. Process-level servers implement extended abstractions using these basic interfaces. Some amount of device support must be provided in the kernel because device interrupts go to the kernel, some device control operations are privileged, and kernel control of some device operations is required for kernel integrity. As an example of the latter, a faulty or malicious process could initiate a disk DMA operation that would overwrite the kernel unless the kernel has control over the DMA controller. Without this control, no guarantees could be made of the kernel's correct operation without verifying all such modules outside the kernel, an unacceptable requirement for reliable and secure systems.

The kernel interface to the mouse illustrates one such minimal interface and the partitioning of function between the kernel and process levels. The mouse appears as an open file that contains the x and y coordinates and the button positions of the mouse. A process reading from the mouse file is suspended until a change has occurred in these values since the last time the mouse file was read. The kernel mouse handling code performs the polling and interrupt handling of the device interface to keep the file data up to date. With this interface, no process activity need result until the mouse moves or has a button change position. However, the normal events associated with mouse changes, including moving the cursor, popping up a menu, and such like are all performed at the process level. The efficiency of the V lightweight process mechanism allows the cursor tracking, rubberbanding and other real-time display functions to be implemented at the process-level with entirely acceptable performance.

As another example, the disk interface provides access to each drive as a raw block device, an array of integer-indexed data blocks. The file server implements files using this interface.

Network connections are handled similarly. For example, the kernel provides a block interface to the Ethernet, providing the ability to read and write raw Ethernet packets. The Internet server at the process level implements TCP/IP, UDP and X.25.

As a final example, a graphics frame buffer is handled as a block device of size corresponding to the memory area of the frame buffer. Using the virtual memory system, the frame buffer can be mapped into the user process's address space and accessed directly. Thus, the (process-level) V display server is able to access the frame buffer with the same efficiency as if it were kernel resident. Because devices use the UIO interface and the virtual memory system binds UIO objects into address spaces, no special provision is required for these types of devices.

Kernel Design Mistakes

The kernel design as presented so far may appear as a straight-forward success story. The reality is that the design has been (and continues to be) an iterative process in which new ideas are tried out and old mistakes are (painfully) thrown out. The following are some examples of "dirty laundry."

The original design structured process identifiers with a "logical host" subfield, which was used to simplify allocation of process identifiers and mapping process identifiers to the right host. However, as pointed out by the work of Theimer et al. [33], this mechanism imposed unreasonable restrictions on the process migration facility because all processes associated with a logical host had to be migrated together. It also led to complexity in the kernel to handle multiple logical hosts per physical host. At the time of writing, this subfield has been eliminated and we are working to get the process migration facility working again after the revisions. In the revised design, an individual process can be migrated although normally one would migrate all processes in an address space along with the address space itself.

The original design also minimized the use of network-level broadcast or multicast. Basic naming was provided by a special purpose *GetPid* function which mapped *logical* process identifiers to actual process identifiers using broadcast[4]. The primary use was to

[4] This function originated in Thoth where it was also used as the basis for the system naming mechanism.

locate a name server. Process identifier allocation used another specialized broadcast mechanism. These mechanisms required highly specialized code in the kernel which was repeatedly found inadequate or incorrect. The introduction of multicast and process groups eliminated these design mistakes from the kernel and the problems with these special-purpose mechanisms. However, the concept of local process groups was introduced at the same time, and this too turned out to be a significant problem. A *local group* is one in which all members of the group are local to one host. Local groups were recognized for the optimization of being able to unicast to the one host to communicate with the group, as opposed to the normal multicast transmission. However, with the introduction of process migration, a process group that started out local to one host could not be guaranteed to stay that way without restrictions on migration or additional complications in the kernel. Moreover, local groups were being used with some contortions to address the managers of particular processes; this use has now been replaced with the co-resident addressing mechanism. The kernel support for local groups has now been removed, simplifying the group management and migration code significantly.

As a final example, process and memory management were originally provided in the kernel by a single server pseudo-process, receiving request messages and replying in the message model of a server. Because the kernel otherwise executes as a shared (but protected) library of procedures invoked by the process level, this server structure required considerable specialized code, exhibited poor performance and suffered from subtle errors. In recognizing that one could easily support the remote procedure invocation model in the basic IPC mechanism, we revised the kernel server invocation to use procedural invocation, thereby eliminating the specialized message handling code for the kernel server. Subsequently, partitioning these services across multiple servers corresponding to process, memory, device and communication management improved the modularity of the kernel.

In general, one great luxury we have in a university research environment is the time to revisit and revise a design that we have made to work. The incorporation of fresh insights into the design allows our understanding of good kernel design to iteratively improve, with real testing of this supposedly improved design at each stage. After all, as a research effort, our role is to build an improved understanding of kernel design, not just an improved kernel. The iterated kernel design also provides the base on which to explore the next level of research ideas, many of which are inspired by, and made possible by, the V kernel facilities.

I/O

Input and output are conventionally regarded as primary services of the operating system, the means by which a program communicates with its environment. In V, a program communicates with its environment, including other programs, using the interprocess communication facilities. The V I/O system is really just a higher-level protocol used with the IPC facilities so that programs, subsystems and modules can interact at a larger grain than individual messages. It imposes a standard structure and interpretation on the contents of the messages that are exchanged. This application-level implementation of I/O contrasts with the conventional approach in which I/O is implemented as a kernel-resident module of significant size and complexity.

A key issue in an I/O system is the uniformity of the interface. An application should be able to bind dynamically to any one of a wide range of I/O services, rather than having to be written specifically for a particular I/O service. This property is particularly important in a distributed system in which extensibility is important and heterogeneity is common. For example, a distributed system may include multiple file servers running different file system software with different file attributes. The challenge is to define a uniform I/O interface that maximizes performance and functionality in the distributed environment across a wide range of I/O-like services.

V uses the UIO interface [10] as its system-level I/O interface (as opposed to the application-level interface, which is implemented by the run-time I/O library in terms of the UIO interface). In the UIO model, I/O is accomplished by creating a *UIO object* that corresponds to an open file in conventional systems. Read, write, query and modify operations are then performed on this UIO object. The UIO interface specifies the syntax and semantics of these procedures; a presentation protocol specifies the mapping of the procedure parameters onto IPC messages, analogous to the calling conventions used by compilers. Programming language I/O operations, such as the C `getc` and `putc` operations, are mapped onto the UIO operations by the run-time libraries for the language.

The UIO interface departs from conventional system I/O interfaces in several ways. First, the UIO interface uses a block-oriented data access model. That is, a UIO object is viewed as a sequence of data blocks that are read or written, rather than a byte stream. The block model supports access to I/O services in which multibyte units have semantic significance, such as arises with network packets, database records and terminal input lines. The block concept is also used in other services to indicate to the client an efficient unit of transfer and buffering for reading and writing. Multiblock reading and writing is also supported by some servers.

Second, the UIO interface is a *stateful* interface. The UIO object represents state that must be initialized prior to other I/O operations, must be reclaimed when no longer needed, and must be recreated for recovery after a I/O manager crash. The stateful interface is required to handle I/O services such as pipes, windows and network connections (to name but a few) which only exist when "open" and are not amenable to the so-called stateless techniques used in WFS [31] and NFS [30]. The stateful interface also provides a mechanism for handling the client I/O state associated with locking and recovery required to support atomic transactions.

Finally, the UIO interface divides functionality into *compulsory, optional* and *exceptional* functionality. The compulsory functionality represents the lowest common denominator, roughly corresponding to a (sequentially accessed) read-only or write-only stream. The optional functionality allows individual I/O services to make extended functionality available when feasible and necessary for the particular service. For example, file service should provide random access, not just sequential access. I/O services indicate extended functionality in the UIO interface using standard attributes of the UIO object. for example, the *STORAGE* and *RANDOM_ACCESS* attributes indicate that the UIO object implements storage[5] and random access respectively. The attributes allow a client to check that the UIO objects it is using have the required facilities for its operation, avoiding discovery by failure at some inconvenient point in its execution. The attributes also allow some important optimizations to be made in the I/O runtime library. Finally, a control function provides a standard escape for invoking specialized I/O operations, such as device-specific operations.

The UIO interface is implemented by a wide range of V services, including files, pipes, Internet protocols, multi-window displays, devices and printers. In addition, several other services use the UIO interface to provide access to directories of information that they maintain, even though the service itself may not fit into the I/O paradigm. For example, the V program manager (or *team server*) implements a directory of the currently executing programs in this fashion.

TABLE II. UIO Reading: Time per Kilobyte on Microvax II (in milliseconds)

Server Location	get-byte	Disk (bytes)	Disk (blocks)	IPC
Local	6.23	9.91	3.47	1.79
Remote	6.23	14.63	8.18	6.34

The cost of the UIO interface for byte-stream and block-stream access is indicated by Table II. The get-byte column lists the basic processing time for calling the V version of the UNIX getc 1024 times (to return 1024 bytes), not including any filling or flushing of the local buffer. (This measurement corresponds to 160.5 kilobytes per second or 6.08 microseconds per byte.) The next column gives the elapsed time per kilobyte to do a 1 kilobyte read using 1024 getc operations (including the cost of reading the 1 kilobyte read from the file server). The third column lists the elapsed time for a 1 kilobyte block read (without getting each byte) from the V file server. The final column indicates the basic interprocess communication cost portion of these operations. If we view the UIO interface overhead as the cost of reading minus the cost of the basic IPC operation as a percent of the total time (factoring out the get-byte overhead in the former case because it should be the same (or worse) in any byte stream I/O implementation), the overhead for the UIO interface is 11 percent for byte stream I/O and 22 percent for block stream I/O for remote I/O. The UIO overhead includes the cost of generating UIO request messages and processing responses at the client end as well as the processing of client UIO requests (including extracting the request data from the buffer pool) at the server end. Because these overhead costs are essentially independent of the amount of data requested, reading in 8-16 kilobyte units reduces this overhead to negligible levels, further supporting the conclusions of a previous report [25] that 8-16 kilobyte reading appears more efficient than larger or smaller sizes of transfer unit.

The UIO interface illustrates another important principle in distributed system design: separation of system-level interface from application interface. The UIO interface is a system-level interface, optimized for performance, reliability, security and flexibility. An I/O run-time library implements the application abstraction in terms of the system-level interface. For example, the get-byte I/O interface provided in C and Pascal is implemented by the run-time library in terms of a local buffer and the UIO block read and write operations to fill and flush this buffer, as was illustrated in Figure 2. The distinction between the system and the application interfaces takes on greater importance in distributed systems than previous centralized systems because of the following:

- The "system call" in a distributed system may entail communication with a remote node and thus incur far greater cost than in a centralized system. Thus, adding function to the run-time libraries to reduce the frequency of remote system calls (thus further separating the application and system interfaces) significantly improves performance.
- Server processing is a critical system resource because servers typically support a large collection of clients. Migrating the processing load from the servers to clients by adding to the run-time routines offloads the shared servers and improves overall system performance.
- The reduced cost of semiconductor memory has all but eliminated the importance of using system service modules as a mechanism for run-time code sharing.

We have taken these considerations into account in designing the UIO interface and plan to explore these issues further in other areas of V run-time support.

NAMING

A number of system operations, such as file open, query and modify, take a character-string name to specify the object on which to act. The system needs to provide character-string naming with flexible user-level name specification, efficient mapping, binding and unbinding plus support for hierarchical structure, including directories and the *current working directory*

[5] A UIO object with the STORAGE attribute guarantees that a block that is reread returns the same data as before unless it has been overwritten in the interim.

feature in UNIX [29]. In addition, a modern system is expected to provide extensibility to new user- and application-defined objects. A distributed system must address this problem, recognizing that the objects may be implemented by many different nodes in the system. The naming problem is not restricted just to character string names; an operating system must provide a way to refer to a variety of different objects, including processes, address spaces, communication ports, and open files using compact binary identifiers. The V naming facility is based on a three-level model, structured as character-string names, object identifiers and entity identifiers.

Character-string Names

In the V design, we observed the most efficient naming design from a communication standpoint is to have each object manager implement names for its own set of objects. For example, each file server implements its own directory system. Then, operations on objects specified by name can be handled directly by the object manager without communication with a name server, provided only the client can determine which object manager to contact, given an arbitrary name. This approach takes advantage of the fact that a name is generally only mapped as part of an operation on the object. For example, a file name is generally only mapped to the file as part of opening the file, removing it or querying or modifying its file attributes. This approach has several significant advantages in addition to efficiency. First, consistency between objects and the directory entries for the objects is simplified because both are implemented by the same server. The design also eliminates the need for a client-visible unique object identifier as an identifier returned by a separate name server and passed to the object manager, as is required in the alternative design. Second, this design results in the object directory being replicated to the same degree as the objects being named, because the directory is replicated when the manager is replicated. Thus, a client never suffers from an object manager being available but effectively inaccessible because of a name server failure. Finally, this design facilitates incorporating "foreign" or independently developed services, which typically already have their own directory system and (sometimes) their own syntax. With the merits of this approach, our work has focused on the design of an efficient, reliable and secure mechanism that ties these individual object manager directories into a system-wide name space and directory system. The result is the V naming protocol which we describe next.

Each object manager mounts its object directory (or directories) into the global name space by picking a unique global name prefix for the object directory and adding itself to the *name handling (process) group*. Uniqueness may be ensured by a human administrator, by a global name server, or by sending to the name-handling group to check for duplicates. A client program can then locate the appropriate object manager for a given character string name by multicasting the *QueryName* operation to the name handling group, as suggested in Figure 6. Only the appropriate server responds.

Each V program maintains a cache of name prefix to object manager bindings that eliminates most of the multicast queries. This cache is initialized on program initiation to avoid startup name cache misses in the common case, similar to the way that environment variables in UNIX are initialized by the shell on program initiation. With this cache mechanism, we measure that only 0.3 percent of the name lookup operations result in a multicast query operation. It should be noted that the use of multicast to locate the object managers means that a name lookup (for a valid name) always succeeds if the network is working and the object manager implementing the object is operational.

A problem with this basic design arises with the mapping of invalid names or names for which the associated object manager is inaccessible. The client simply does not receive a response to its multicast query and thus cannot determine whether the name is invalid or simply inaccessible at this time. To address this problem, we combine our decentralized approach with a highly resilient global naming system, as described by Lampson [22]. The decentralized approach provides efficient, resilient name mapping for performance-critical operations such as file opens. The global directory mechanism provides a highly available database indicating which portions of the global name space actually correspond to object managers. It also can be used to avoid multicasting "globally" on a name cache miss.

Several other issues arise with this design, including efficient handling of current working directories, implementation of name aliases, handling of object directories partitioned across multiple servers and detection of counterfeit name query responses. The interested reader is referred to a forthcoming report [15] for further discussion of this work.

One can view the V directory system as implementing a shared memory to store the name bindings using caches and multicast, analogous to the techniques used in a shared-memory multiprocessor machine. The servers provide primary site storage for the bindings while the client name caches correspond to per-processor caches. The major difference is the way in which consistency is handled. Multiprocessor machines rely on an efficient, reliable broadcast facility at the hardware level plus a write-broadcast or ownership protocol [2]. The V name caches rely on *on-use* detection of stale data, relying on the fact that names are only mapped as part of an operation invoked at the object manager. For example, a client may discover that a cache entry is stale when it uses the entry to map a file name to a particular server. It then deletes this cache entry and uses the multicast query name operation to get an up-to-date entry.

Exploiting problem-specific characteristics to maintain consistency dramatically reduces the cost of consistency maintenance for name caches compared to that required to guarantee strict consistency. Moreover,

placing the name cache in each program's address space makes conventional consistency maintenance (across all executing programs) infeasible but makes the name cache access efficient. We have observed and exploited similar benefits in a number of other situations, including distributed scheduling, time synchronization, atomic transaction management and distributed game programs. *Problem-oriented shared memories* [8] of this nature appear to have general applicability in distributed systems as a compromise between shared memory with its conceptual appeal but high consistency cost over unreliable networks and ad hoc communication techniques with efficiency benefits but significant programming complexity.

Object Identifiers

Operations such as a file open map a character-string name to an object; this object is a UIO in the case of file *open* operation. An efficient *object identifier* is used to refer to the object in subsequent operations, avoiding the overhead of character-string handling and lookup each time. V Object identifiers are structured as shown in Figure 11. The manager identifier is an IPC identifier that specifies the object manager or one of its ports that implements the object. The local-object-id specifies the object relative to this object manager. Object identifiers are used to identify open files, address spaces and contexts or directories.

manager-id	*local-object-id*

FIGURE 11. Object Identifier Structure

With this structure, mapping an object identifier to its implementation is efficient because the embedded transport-level identifier for the object manager can be used by the client to efficiently access the correct manager module. The manager module then uses manager-specific mechanisms to map the second portion of the identifier to its implementation. Allocation of object identifiers is also efficient because the manager module can allocate a new unique identifier without interaction with other managers, the uniqueness being conferred by prepending its entity identifier. A similar argument applies for deallocation.

Object identifiers are only used to identify objects whose lifetime does not exceed the lifetime of the service entity identifiers, because the entity identifier is invalidated when the process crashes. Also, an object manager is assigned a new entity identifier on reboot. This approach also avoids going to long identifiers to effectively guarantee against reuse, as would be required if the identifiers were used for long-term objects such as files. For instance, in such a system, a user could present such a file identifier at some arbitrary time after the file had been deleted. The file server would have to avoid reusing this identifier to avoid the confusion that would otherwise ensue. Instead, object identifiers are used in V for transient objects such as open files. As mentioned earlier, permanent objects such as files are named using character-string names.

An object manager may be replicated or distributed across multiple nodes. For example, processes as objects are implemented by the distributed process server in the kernel, with a server instance on each node handling the processes currently local to that node. Similarly, a replicated file system maintains replicas of each file on each of several file server nodes. In both cases in V, the object manager as a whole is identified by a group identifier for the group of server instances, one per node, with separate individual identifiers for each instance. In the distributed case, a client may use the group identifier to identify the server handling the particular object of interest and subsequently use the instance identifier for the particular server, avoiding multicast to the group on every operation. For example, a client performing an operation on a process can locate the particular server handling the process using the group mechanism and then use the identifier for that server for subsequent operations. When the object migrates or the specific object manager crashes, the client receives an error message on use of the individual server identifier and rebinds to the new instance using the group identifier. For example, a client reading a replicated file in the *read-only* mode defined in the UIO interface [10] can rebind to another object manager supporting another replica if the specific object manager it is using fails. Similarly, when a process is migrated, the kernel process server identity for the process changes. A client operating on this process has to rebind to the new process server. Alternatively, when the desired server is determined by its co-residency with a given process, the co-resident addressing can be used in conjunction with the group address to address the particular server. This mechanism works with the same efficiency as when the individual server is addressed directly; it also automatically rebinds (at the IPC level) when a process migrates or a manager crashes.

The group addressing is also used for operations that affect the entire replicated or distributed manager. For example, a write to a replicated file uses the group address to update every copy. The client uses its list of the individual servers and the responses it receives from these individuals to ensure that every replica acknowledges the write operation.

This discussion should make it apparent that the kernel IPC naming, the process and process group identifiers, provide the basis for character-string naming and object identification. This third and lowest level of naming is discussed next.

Entity, Process and Group Identifiers

Entity identifiers are fixed-length[6] binary values used to identify processes, groups of processes and transport-

[6] Entity identifiers are currently 32 bits. However, we are changing to use 64-bit identifiers.

level communication endpoints. The entity identifier is used with the V communication primitives to identify transport-level endpoints. Entity identifiers have also served in V as process identifiers because a process effectively has a single logical port on which to send and receive messages.

A key property of entity identifiers that distinguishes them from the endpoint identifiers in other transport layer facilities is that they are host-address independent. That is, a process can migrate from one host to another without changing entity identifiers. This host independence requires large entity identifiers as well as a mapping mechanism from entity identifiers to host addresses. The V kernel maps entity identifiers to host addresses using a cache of such mappings along with a multicast mechanism to query the other kernels for mappings not found in the cache, analogous to the name mapping cache described in the section on Character-string Names. Group identifiers are mapped using an embedded subfield in each identifier that is hashed to a base multicast address used by V to generate the multicast host address for the group. Thus, group identifiers are mapped many-to-one onto a range of multicast addresses (or host group addresses [13]). Host addresses are network- or internetwork-dependent and handled by a low-level module in the kernel, rendering most of the kernel and process-level V software network-independent.

Another difficulty with host-address independent identifiers arises with allocation because guaranteeing uniqueness requires cooperation among all instantiations of the kernel. Moreover, to avoid confusion, the kernels must cooperate to ensure that an identifier cannot be reused too quickly after its last use[7]. Otherwise, the meaning of the identifier (which process it binds to) may change over time, leading to incorrect behavior by users of the identifier over this time period. In the original V design (coming from Thoth [9]), there were simple mechanisms that attempted to provide what we call *T-stability*—an identifier does not get reused in less than T seconds. In redesigning the protocol for more general Internet use, we have gone to 64-bit identifiers to reduce the expected frequency of reuse. The difficulty of implementing T-stability in a distributed environment was not sufficiently recognized in an earlier report [12] and is the subject of further investigation.

V SERVICES

The V kernel facilities, the naming protocol and the UIO interface provide a basic framework for implementing a variety of services. A number of service modules have been designed and implemented that are of research interest in their own right.

These service modules share a number of attributes in common. First, they are structured as multiprocess programs, exploiting the lightweight processes provided by the V kernel. Second, most of them implement the V naming protocol and UIO interface, the latter either because they implement open file-like objects or because they implement an object directory that is accessed using the UIO interface. Finally, client access to their services is provided entirely through the V IPC primitives. (We are currently working to extend some services to support the V atomic transaction protocol as well.)

The pipe server implements UNIX-like pipes outside the kernel using the IPC primitives and the UIO interface. A pipe provides the "sex matching" that allows two clients to connect with the asymmetric interconnection provided by the UIO interface. In addition, pipes support buffering, multiple readers and multiple writers. Zwaenepoel studied this server [34] to investigate the performance penalty from using a process-level server as opposed to a kernel-level implementation. Measurements indicate the penalty is under 15 percent using 1 kilobyte data blocks. We have found the performance adequate for our uses of pipes, especially given that high performance IPC to servers is provided directly by the normal IPC primitives. Pipes are used primarily for relatively modest amounts of interprogram I/O activity.

The Internet server implements the TCP/IP suite of protocols [27] using the basic network access provided by the kernel device server. Like the pipe server, the Internet server relies on the V kernel for lightweight processes, real-time scheduling, accurate timing and fast interprocess communication to achieve good performance without compromising its modular, multiprocess structure. Lantz et al. [24] report on the performance of some applications using this server. The performance of this service is competitive with performance reported for the UNIX kernel implementation of TCP and the benefits of implementing this service outside the kernel are considerable. Besides allowing the kernel to be much smaller, the Internet server has been much easier to develop, debug and maintain than if we had done a kernel implementation. In addition, the Internet server is loaded on demand in V rather than permanently configured in the standard system. Finally, it is not uncommon for a remote terminal connection to execute with the terminal program local to the workstation but with the Internet server running on a second machine, possibly sharing the server with other clients. Because of the fast V IPC, the performance difference using a remote Internet server rather than a local instance is not generally noticeable, even for character echoing.

The V file server was derived from the Thoth file system and uses the same file descriptor and block allocation disk data structures [9]. Most of our work with this module has focused on providing a buffering scheme that is well adapted to using large amounts of RAM. In particular, the buffer pool currently uses 8 kilobyte buffers (which can be made larger), allowing large network and disk transfers with minimal overhead. (The contiguous allocation scheme of the file system results in most files being data contiguous on the

[7] The finite size of entity identifiers and their dynamic allocation makes reuse necessary and inevitable.

disk even though the block allocation unit is 1 kilobyte.) Preliminary performance figures indicate significant benefits from this approach [16]. We have also been exploring the multi-process structuring of the file server with the goal of achieving efficient parallel execution on the multiprocessor machines to which we are porting V.

The printer server, developed by Tim Mann, exhibits several interesting properties, even though it was never a research project per se. First, it supports spooling of print jobs even though it runs on a diskless node, exploiting V network IPC to write files to network file servers. (A new recently installed configuration supports local spooling of files using a disk and a local instance of the file server.) Second, it supports multiple client protocols, allowing print files to be submitted using either V IPC and the UIO interface or using TCP connections. The latter access is implemented by the printer server running an instance of the Internet server.

The team server[8] or program manager handles the execution of programs on its host machine. It provides an interface between client programs and the kernel when initiating the execution of a program. It also implements time-slice scheduling of programs with foreground, background and "guest" priority classes. In addition, it serves as a process-level exception handler, invoking the interactive debugger on faulting programs. It also maintains a real-time database of information on programs in execution and resource consumption statistics for programs and the host itself. Using this information, it participates in the distributed scheduling of programs within the cluster of machines constituting a V domain. Finally, it handles program termination and assists with process migration (although most of the logic is handled by a separate program). We plan to further exploit this process-level module to explore a variety of issues in distributed scheduling both of single node as well as multi-node distributed parallel programs.

The V display server implements multi-window facilities using a bitmap display, a commonplace facility in modern workstation systems. Like the other servers, it makes good use of the processes and efficient interprocess communication. For instance, mouse tracking is performed by a helper process that sends updates to the display server to reposition the cursor. The display server also represents an early effort to provide a high-level graphics representation at the client interface. This high-level representation significantly reduces the data rates for transmitting structured data. More importantly, it allows some operations to be performed local to the display server, rather than relying on application facilities. For instance, the display server supports multiple views, zooming and redraw, making these facilities available for all applications. Further details on this work are described by Lantz and Nowicki [23].

Work continues on new servers, including a log server for optical disk [20], an atomic transaction server and a time synchronization server.

[8] The term *team* is used in *V*, as it was in Thoth, to refer to a set of processes sharing the same address space.

THREE CLASSES OF V APPLICATIONS

Operating systems of the past have been generally targeted for (interactive) timesharing, batch processing or real-time control. V ambitiously attempts to handle all three classes of applications.

The multi-user workstation cluster, illustrated in Figure 12, is the distributed systems equivalent of the conventional interactive timesharing system. It differs in that a user's workstation provides most of his processing resources in addition to display, keyboard and mouse. The backend hosts or mainframes are reduced to serving primarily (or possibly, exclusively) as file servers and computation servers. Using the V system, each node runs a copy of the kernel plus various server programs. Each node with secondary storage may run the V file server software and offer file service. The kernel's interprocess communication makes this service and others available in a network-transparent fashion to all nodes on the network.

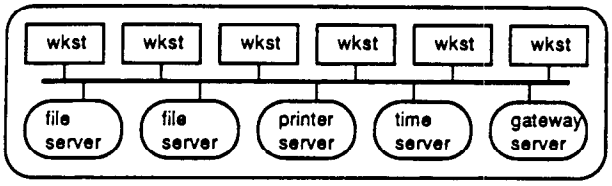

FIGURE 12. Multi-user Workstation Cluster

V also deals with the issue of the processing resources of the cluster being fragmented across the processors of the nodes, instead of being concentrated in the single processor of a conventional timesharing system. In particular, a user can transparently run a program on another node in the cluster to make use of available computing cycles. In practice, we see no perceptible difference even running character-at-a-time interactive editors such as *emacs* remotely on another workstation. Using an experimental scheduler, V runs each program on the least loaded node in the cluster, thereby automatically distributing the load. With a workstation per person, we observe a very low average load per workstation, similar to what has been observed for telephones and personal automobiles. As a consequence, pooling the resources of the workstation cluster is surprisingly effective, eliminating the need to dedicate a pool of processors as computation servers, at least in our environment. Besides saving on the base hardware investment, this sharing of resources makes the latest and fastest workstations in the cluster available to everyone.

Several aspects of V are essential to make this facility practical besides fast, network transparent interprocess communication. First, the kernel encapsulates a program in an address space so that it is no more a threat to other programs when run on the same machine than when it is run on a different machine. Thus, a user

need not fear that guest programs will crash his machine or damage his programs or storage. Second, the V scheduler runs guest programs at lower priority to minimize the interference they can cause local programs. This priority affects all aspects of a program's execution including access to the processor(s), network interface, servers, etc. Thus, the primary point of contention is on the use of physical memory. We do not expect this contention to be significant for systems with large amounts of memory, as expected in the future. Finally, V provides the ability to migrate a running program to another node, allowing a user to offload guest programs entirely. This also allows the load from long running programs to be redistributed.

With these facilities, a workstation cluster can have all the advantages of a centralized timesharing system, including shared file system, shared processing resources and multi-user community services. In fact, the total processing capacity often far exceeds that of many current timesharing systems. For example, a cluster of 25 4-megabyte Digital Microvax II workstations is roughly 25 MIPS of processing power with a total of 100 megabytes of memory. This configuration is far less expensive than a conventional mainframe of comparable capacity. It also provides better interactive support (including bitmap display and mouse) and it almost never *completely* crashes. The major potential disadvantage of the workstation cluster is the difficulty in harnessing a significant portion of the 25 MIPS to work on one program.

Distributed Parallel Machine

Modern workstations such as the Digital MicroVAX, the Apollo and the SMI SUN provide cost-effective computation power in the 1-10 MIPS range and are destined to get cheaper and faster, benefiting from the economies of mass production and VLSI technology. A cluster of such machines would be a cost-effective way to configure a powerful computation engine if only one could write programs that could make good use of the computational resources in the form provided, namely, multiple processors and no physically shared memory.

Some recent work by Michael Stumm and myself [18] is directed at understanding how to structure programs for this environment and investigating the adequacy of the V facilities for such distributed parallel programs. We structured several programs in what we call the *multi-satellite star* model, logically depicted in Figure 13. The application is structured as follows:

- A set of application-level instruction sequences we call *subtasks.*
- A *satellite* processing module that executes subtasks.
- A master module, called *star central*, that allocates subtasks to satellite processors, generates new subtasks based on the results of other subtasks and detects sufficient conditions for termination.

We have programmed several example problems using this model, including the traveling salesman problem, alpha-beta search, zero-finding and matrix multiplication. In each case, we found the problem was easily

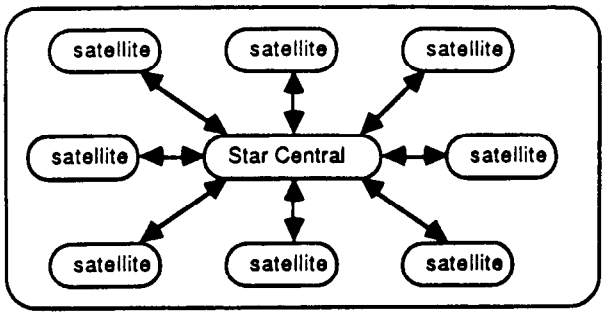

FIGURE 13. Multi-satellite Star Computation

programmable in the multi-satellite star model and significant performance benefits were achieved over using a single processor. Although the speedup for some programs, such as matrix multiplication, did suffer from the communication overhead, the major significant problem was the superfluous processing that often arises as a result of parallelizing the computation. We also observed the V multicast facility was a useful way to distribute intermediate results within the computation, cutting down on this extra processing to some degree.

We conclude from this preliminary study that a workstation cluster running V has much the same computational power for many problems as a shared memory multiprocessor. The key issue appears to be understanding how to program applications to execute in parallel, with the differences between a shared memory and distributed parallel machine less significant, at least for many applications. Overall, it appears feasible to extract a considerable amount of the latent processing power in a workstation cluster for heavy duty computation. Additional understanding of parallelism and language support is required. The V operating system facilities seem adequate although further improvements in the network interprocess communication performance would be of benefit.

The star central node serves in part as a form of shared memory in which the global state of the parallel computation can be maintained. We are experimenting with the distributed shared memory provided between nodes by the virtual memory system, as described in the earlier section on Memory Management. Judging by the experience reported by Li [26], we expect this approach to be applicable to a significant class of distributed parallel programs, providing shared memory similar to that available in a shared memory multiprocessor, differing primarily in the performance penalty for contention.

We originally considered having a pool of dedicated computation server machines to support remote and distributed parallel computation—a so-called *processor pool.* However, we found the utilization of the workstation resources with a workstation per person to be sufficiently low that additional processors were not required. Moreover, we observe that each new generation of workstations is so much faster than the previous generation that a dedicated pool of previous generation

processors would likely get less use than a few idle workstations of the current generation. In essence, these observations point out the merit of software making good use of the current generation of hardware so that the administration of the computing cluster can save its money for the next generation of hardware. To allow for this evolution as well as accommodating heterogeneity within one generation, V handles program execution with different processor types and machine configurations. Currently, the two major types of architectures it handles are the VAX and the SUN. The VMP machine [17] represents a third type.

Distributed Real-time Control

A third class of applications for V is real-time control. A distributed implementation of a real-time control system has the well-known advantages of extensibility, cost-effective performance, reliability and security. However, it also has the advantage of allowing each node to be physically co-located with the equipment it is monitoring and controlling even though the equipment may be physically distributed, such as in a factory or a battleship. This co-location minimizes communication requirements, simplifies equipment placement and configuration and improves reliability and security in dangerous environments.

A distributed system also provides multiple processors so that there can be, in the extreme, one processor for each sensor or actuator, eliminating the scheduling contention and scheduling algorithm complexity required for real-time response in centralized single-processor real-time systems. This configuration is illustrated in Figure 14. The primary problem for a distributed real-time system is that of maintaining the shared state of the system across multiple nodes within the real-time requirements of the system.

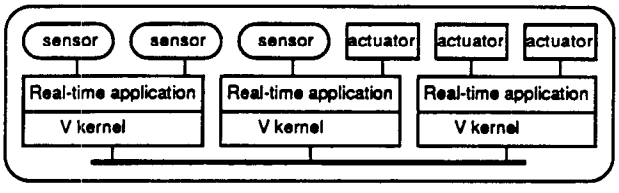

FIGURE 14. Distributed Real-time Control

Several extensions to the interprocess communication in V support efficient distributed state update. First, V supports a datagram message as a degenerate form of message transaction. Combining this facility with multicast, a process can send out periodic updates to the other controlling nodes in the system without blocking for retransmission, timeout or waiting for a response. A single multicast datagram thus updates all other nodes with high probability at the cost of a single transmission to the updater, as shown in Figure 15. The recipients of these datagram updates can notice when they fail to receive an update from a particular node for some time and explicitly request an update or take other corrective action. However, receipt of a subsequent datagram update normally compensates for the loss of a previous datagram. We have implemented this technique successfully using V in the context of a distributed multi-player game program [4] as well as in a student project which implemented control of a (simulated) robot arm. In addition, Tektronix has been using V as the basis for distributed instrumentation. We were recently given a demonstration of a distributed oscilloscope with the display separated from the sensor by an Ethernet, with the V IPC providing communication between the sensor program and the display program.

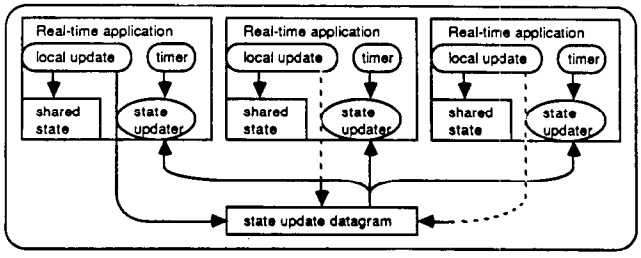

FIGURE 15. Distributed Real-time Update

In addition to datagrams, V also provides prioritized message transmission and delivery and conditional message delivery.[9] Of course, V also has strict priority-based scheduling, accurate time services and memory-resident programs, the other key requirements for supporting real-time applications.[10]

Although the basic techniques we have described are not deep or even novel, the ability to run such applications on top of a general-purpose operating system kernel is a departure from previous practice. We believe the increasing power of processors and networks and the improved understanding of the key operating system services allows general-purpose distributed systems such as V to serve as the base for a wide variety of most, if not all, real-time applications. The benefits of generality are great.

Although these three classes of applications have been described separately, there is no reason that V cannot support all three concurrently on the same hardware configuration, provided that sufficient hardware resources are available. For instance, strict priority scheduling and resident (non-paged) memory allocation allows real-time processes to run independent of lower priority user and compute-bound processes. Even network access and message delivery are prioritized.

This integration seems appropriate for the factory of the future where real-time control of the factory floor, office processing, and simulation of manufacturing processes and schedules are all computerized. Sharing the same hardware base for all activities reduces the

[9] Conditional message delivery means that the message is delivered only if the receiver is awaiting a message when the message arrives.

[10] Deadline scheduling can be accomplished by dynamic manipulation of process priorities.

hardware cost for the required performance and reliability, guards againsts artificial information and functionality barriers that can arise in less general systems and provides for greater extensibility and reconfigurability. We hope to explore this application area in the future.

PRESENT STATUS AND FUTURE PLANS

The V software has reached a reasonable level of utility and maturity.[11] It is being distributed under license by Stanford[12] and is in use at several other universities, research laboratories and companies. After several years of intense experimentation and extensions, we are engaged in a significant effort to revise and rebuild the system to correct design mistakes, improve the quality of the implementation and incorporate new insights. This investment is justified because V is a vehicle that allows us to explore research territory with far less effort than starting afresh building a system with each new project and research direction. The scale and maturity of V also provides our research with far greater credibility than work lacking experimental evaluation. Moreover, it allows our ideas to be incorporated into a system in daily use, giving strong feedback on the real utility, efficiency and resiliency of these ideas in practice.[13] We expect to have this reimplementation effort reflected in the distribution of V software by this summer.

There are several major directions in which research with V is progressing. First, we are interested in studying the operating systems issues in supporting parallel and real-time applications on shared memory multiprocessor machines. Operating systems of the future should accommodate multiple processors with the same ease with which they currently accommodate (for example) multiple disk drives. To this end, we have modified V to run on a shared memory multiprocessor machine in a fully symmetric fashion. The target machines include the DEC experimental Firefly multiprocessor workstation [32] and VMP [17], a shared memory multiprocessor machine we have designed and built.

Further, we are exploring a number of aspects in the area of computer communication, most of which are direct outgrowths of our experience with the V distributed system. We are designing a high-performance network interface to improve the performance of interprocess communication, with particular focus on the 100- to 1000-megabit networks of the future. Another project is developing a transport-level gateway that insulates the local cluster from the performance, reliability and security complexities of wide-area networks and protocols.[14] Besides allowing local protocol implementations to be optimized for local communication, this approach provides a *firewall* between a cluster and the rest of the world. V is serving as a real-time kernel on which to implement this gateway as well as providing a local network protocol.

We are also attempting to export some of the V protocols into the computing community. There is a project to extend the DARPA Internet to support multicast [13]. In addition, we have been working to refine VMTP [12] into a protocol suitable for use as a general-purpose request-response (RPC) protocol.[15] We hope to offer a naming and an I/O protocol to the community in a similar fashion. More generally, we see the need for a standard distributed systems network architecture with a suite of protocols covering the functionality discussed here. We believe the V system, its protocols and their interrelation, have a significant contribution to make to the development of this network architecture.

Finally, we are interested in the problem of distributed information management: how to provide transparent access to structured and distributed information in an efficient, reliable and secure fashion. This problem has many aspects. We have a project to understand how to provide an efficient general-purpose logging facility using the optical disk [20]. We are also experimenting with a distributed atomic transaction management protocol that attempts to make good use of multicast for efficient transaction commit as well as transaction logging. The UIO interface [10] defines some approaches to structured file access, replication and locking to complete the picture. We are currently extending the V file server software to support these extended facilities, including atomic transactions and replication. Finally, we have been investigating approaches to caching structured information using the file caching mechanism and virtual memory system including the file server directories and database views that fit into the UIO model.

These research directions are much easier to explore given that our research group has V at its fingertips.

CONCLUSION

V has been a tremendous learning experience for our research group as well as our students. From the basic tenets given in the introduction, we have evolved a working system and, in doing so, refined and extended the design and our understanding of distributed systems. There are several points that other system designers should consider key aspects of this research and the V design.

First, we focused on the performance of interprocess communication as a key issue. The performance level we achieved with the V IPC is critical to the current system's utility. Moreover, every improvement in per-

[11] This paper was written and formatted in draft form entirely using the V system.

[12] Contact: Office of Technology Licensing, Stanford University, Stanford, CA 94305 for licensing and distribution information.

[13] Readers with experience with large systems will recognize that there is a significant cost to maintaining the approximately 200,000 lines of source codes that constitute this system.

[14] The basic design is described in an early paper [6].

[15] A VMTP protocol specification and Unix 4.3 BSD kernel implementation are available from the Stanford Office of Technology Licensing.

formance extends the range of application of V (especially in the real-time control arena) as well as making the current V applications, such as compilations, run faster. The potential for significant improvements in communication performance using faster networks, high-performance intelligent network interfaces and further protocol and kernel refinements makes the possibilities for distributed systems structured along the lines of V exciting.

Another hypothesis was that the protocols and interfaces, not the software modules, define the system. Thus, we have focused in our work, and in this article, on the design of protocols for data transport, naming, I/O, atomic transactions, remote execution, migration, time synchronization, etc. The implementation of these protocols and their use by a diversity of applications over a period of years (for the more mature protocols) has lead to considerable refinement of the designs. The result is a set of protocols which we believe provide a basis for standardization, not just concepts worthy of further exploration. While this focus on protocols and interfaces may appear obvious to those involved in computer communication, it seems to be lost in many distributed systems efforts in the push to develop software. Ideally, the distributed systems research community should focus on the design and understanding the protocols and interfaces for distributed systems. The commercial software world can then focus on the production of high-quality software that implements these protocols and interfaces, taking confidence from the research work that the resulting modules and systems will meet performance, reliability, security and functionality requirements.

Third, we held the hypothesis that a distributed kernel could provide a base for distributed systems, analogous to that provided by a backplane/chassis for hardware systems. Based on our experience with the V kernel, this approach is extremely successful. Construction of a distributed system given such a base turned out to be much easier than we had originally anticipated. In fact, some of our students are disappointed that there are not more distributed systems issues in the servers and the commands for V. For example, the file server software design is far more affected by considerations of large RAM buffer pools and provision for parallelism than handling of remote clients. However, the design of the kernel itself appears to be a difficult challenge. We continue to have inspirations leading to improvements that make some previous aspect of the design look naive and flawed.

A underlying philosophy of our work was that performance was of paramount importance. No one will use a slow system, independent of its elegance. In exploring this direction, we were surprised at the intellectual challenge presented by performance. We were also surprised at the ease with which we could take a very fast design (once discovered) and package it in a form that is acceptably "clean" for application programmers. We now conjecture that, for every fast design, there exists an acceptably elegant design with comparable performance. That is, one need not significantly sacrifice elegance for performance. However, performance has to be a driving consideration behind the design. In this vein, an unfortunate amount of the work on protocols today is dominated by standards efforts that place performance as one of the last considerations to be addressed. While performance in the slow networks of yesteryear may have been secondary, the multi-hundred megabit, if not gigabit, networks of tomorrow make protocol processing overhead the communication bottleneck for years to come, in spite of increasing speeds of processors. While some may argue that these order of magnitude improvements in communication capacity are not needed, there appears no precedent in the history of computer systems of "unneeded capacity." In fact, these quantitative leaps in computing and communication capacity have historically resulted in qualitative advances in our computing environment. We expect the next generation of computer communication systems and distributed systems to have a comparable effect.

In summary, we have invested considerable time, money and effort in developing an experimental distributed operating system to the point that we can use it for getting our work done, and we continue to pay dearly to maintain this work environment at this level. Nevertheless, the price is well worth it. The process of convincing a cluster of 50 computers to implement a design and subsequent stress testing of the design over periods of months of use have done much to separate the wheat from the chaff in our thinking. The feedback we have received from using the system and from other users of V has also been helpful and stimulating. We plan to push V into new areas of research as long as we have new ideas and the system continues to facilitate their exploration.

Acknowledgements. The on-going support of the Defense Advanced Research Projects Agency has been central to the success of this project. Through their continued support and that of Digital Equipment Corporation we have been able to furnish every member of the project with a workstation and support the significant software development required for this research. Further support has come from the National Science Foundation, ATT Information Systems, Bell-Northern Research, Philips Research, NCR and IBM. We are extremely grateful to Bob Taylor of the Digital Equipment Corporation Systems Research Center for providing access to their experimental Firefly multiprocessor workstations in support of our work in parallelism.

The authorship of the referenced research papers is indicative of the large number of participants in this effort and the contributions of many individuals. Without attempting to do justice to their individual contributions, the participants in the Distributed System Group include (in alphabetical order): Lance Berc, Eric Berglund, Per Bothner, Pat Boyle, Kenneth Brooks, Peter Brundrett, Tom Davis, Steve Deering, Ginger

Edighoffer, Gus Fernandez, Ross Finlayson, Linda Gass, Steven Goldberg, Hendrik Goosen, Cary Gray, Kieran Harty, Zygmunt Haas, Bruce Hitson, Jorge Juliao, David Kaelbling, Hemant Kanakia, Keith Lantz, Chris Lauwers, Will Lees, Tim Mann, Thomas Maslen, Tony Mason, Rob Nagler, Neguine Navab, Bill Nowicki, Erik Nordmark, Joe Pallas, Rocky Rhodes, Paul Roy, Jay Schuster, Andy Shore, Gert Slavenburg, Michael Slocum, Ed Sznyter, Michael Stumm, Omur Tasar, Steve Tepper, Marvin Theimer, Carey Williamson, Michael Wolf, Chris Zuleeg and Willy Zwaenepoel.

REFERENCES

1. Almes, G. The impact of language and system on remote procedure call design. In *Proceedings of the 6th International Conference on Distributed Computer Systems* (Cambridge, Mass., May 19–23). IEEE Computer Society, Los Angeles, Calif., 1986, pp. 414–421.
2. Archibald, J., and Baer, J.L. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Sys. 4*, 4 (Nov. 1986), 273–298.
3. Baskett, F., Howard, J.H., and Montague, J.T. Task communication in DEMOS. In *Proceedings of the 6th Symposium on Operating System Principles* (Purdue University, W. Lafayette, Ind., Nov. 16–18, 1977). ACM, New York, 1977, pp. 23–31.
4. Berglund, E., and Cheriton, D.R. Amaze: A multiplayer computer game. *IEEE Software 2*, 3 (May 1985), 30–39.
5. Brinch Hansen, P. The nucleus of a multiprogramming system. *Commun. ACM 13*, 4 (Apr. 1970), 238–241, 250.
6. Cheriton, D.R. An experiment using registers for fast message-based interprocess communication. *Op. Sys. Rev. 18*, 4 (Oct. 1984).
7. Cheriton, D.R. Local networking and internetworking in the V-system. In *Proceedings of the 8th Symposium on Data Communication* (North Falmouth, Mass., Oct. 3–6). IEEE/ACM, Los Angeles, Calif., 1983, pp. 9–16.
8. Cheriton, D.R. Problem-oriented shared memory: A decentralized approach to distributed system design. In *Proceedings of the 6th International Conference on Distributed Computer Systems* (Cambridge, Mass., May). IEEE Computer Society, Los Angeles, Calif., 1986, 190–197.
9. Cheriton, D.R. *The Thoth System: Multi-process Structuring and Portability.* Elsevier Science Publishers, New York, N.Y., 1982.
10. Cheriton, D.R. UIO: A uniform I/O interface for distributed systems. *ACM Trans. Comput. Sys. 5*, 1 (Feb. 1987), 12–46.
11. Cheriton, D.R. *Unified management of memory and file caching using the V virtual memory system.* Tech. Rep. STAN-CS-88-1192. Dept. of Computer Science, Stanford University, 1988. Also submitted for publication.
12. Cheriton, D.R. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of SIGCOMM 86*, Stowe, Vt. (Aug. 5–7). ACM, New York, 1986.
13. Cheriton, D.R., and Deering, S.E. Host groups: A multicast extension for datagram internetworks. In *Proceedings of the 9th Symposium on Data Communication* (Whistler Mountain, B.C., Sept.). IEEE Computer Society and ACM SIGCOMM, Los Angeles, Calif., 1985.
14. Cheriton, D.R., Malcolm, M.A., Melen, L.S., and Sager, G.R. Thoth, A portable real-time operating system. *Commun. ACM 22*, 1 (Feb. 1979), 105–115.
15. Cheriton, D.R., and Mann, T.P. Decentralizing: A global naming service for efficient fault-tolerant access. *ACM Trans. Comput. Syst.* (1988), to appear. An earlier version is available as Tech. Rep. STAN-CS-86-1098, Computer Science Dept., Stanford University, April 1986, and as Tech. Rep. CSL-TR-86-298.
16. Cheriton, D.R., and Roy, P. Performance of the V storage server: A preliminary report. In *Proceedings of the ACM Conference on Computer Science* (New Orleans, La., Mar.). ACM, Baltimore, Md., 1985.
17. Cheriton, D.R., Slavenburg, G., and Boyle, P. Software-controlled caches in the VMP multiprocessor. In *Proceedings of the 13th International Conference on Computer Architectures* (Tokyo, Japan, June). ACM SIGARCH and IEEE Computer Society, Los Angeles, Calif., sponsor, 1986.
18. Cheriton, D.R., and Stumm, M. *Multi-satellite star: Structuring parallel computations for a workstation cluster.* In *Distributed Computing*, 1988. To appear.
19. Cheriton, D.R., and Williamson, C. Network measurement of the VMTP request-response protocol in the V distributed system. In *Proceedings of SIGMETRICS 87* (Banff, Canada). ACM, New York, 1987.
20. Finlayson, R.S., and Cheriton, D.R. Log files: An extended file service exploiting write-once storage. In *Proceedings of the 11th ACM Symposium on Operating System Principles* (Austin, Nov.). ACM, Baltimore, Md., 1987, pp. 139–148.
21. Kernighan, B.W., and Ritchie, D.M. *The C Programming Language.* In *Prentice-Hall Software Series*, Prentice-Hall, N.J., 1978.
22. Lampson, B. Designing a global name service. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing* (Calgary, Canada, Aug. 11–13). ACM, New York, 1986, pp. 1–10.
23. Lantz, K.A., and Nowicki, W.I. Structured graphics for distributed systems. *ACM Trans. Graph. 3*, 1 (Jan. 1984), 23–51.
24. Lantz, K.A., Nowicki, W.I., and Theimer, M.M. An empirical study of distributed application performance. *IEEE Trans Softw. Eng. SE-11*, 10 (Oct. 1985), 1162–1174.
25. Lazowska, E., Zahorjan, J., Cheriton, D., and Zwaenepoel, W. File access performance of diskless workstations. *ACM Trans. Comput. Syst. 4*, 3 (Aug. 1986), 238–268.
26. Li, K., and Hudak, P. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, (Calgary, Aug.). ACM, New York, 1986, pp. 229–239.
27. Postel, J.B. Internetwork protocol approaches. *IEEE Trans. Commun.* (Apr. 1980).
28. Rashid, R., and Roberston, G. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Asilomar, Calif., Dec. 10–12). ACM, New York, 1981, pp. 64–75. ACM, Baltimore, Md.
29. Ritchie, D.M., and Thompson, K. The UNIX timesharing system. *Commun. ACM 17*, 7 (July 1974), 365–375.
30. SUN Microsystems. *Network File System Specification.* SUN Microsystems, Mountain View, Calif., 1985.
31. Swinehart, D., McDaniel, G., and Boggs, D. WFS: a simple file system for a distributed environment. In *Proceedings of the 7th Symposium on Operating Systems Principles.* 1979.
32. Thacker, C. The Firefly multiprocessor workstation. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Oct.). ACM, New York, pp. 164–172.
33. Theimer, M.M., Lantz, K.A., and Cheriton, D.R. Preemptable remote execution facilities in the V-System. In *Proceedings of the 10th Symposium on Operating System Principles*, ACM SIGOPS, 1985.
34. Zwaenepoel, W. Implementation and performance of pipes in the V-System. *IEEE Trans. Comput. C-34*, 12 (Dec. 1985), 1174–1178.

CR Categories and Subject Descriptors: D.4.7 [Operating Systems: Organization and Design—*distributed systems*; *real-time systems*; *interactive systems*; D.4.4 [**Operating Systems**]: Communications Management; C.2.2 [**Computer Systems Organization**]: Network Protocols—*protocol architecture*; C.2.4 [**Computer Systems Organization**]: Distributed Systems—*distributed applications*

General Terms: Design, Experimental, Performance

Additional Key Words and Phrases: operating system kernel, network transparency, distributed kernel, workstations, interprocess communication, light-weight processes

Received 3/87; revised 10/87; accepted 1/88

Author's Present Address: David Cheriton, Stanford University, Computer Science, Bldg. 460, Room 422, Stanford, CA 94305-6110.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.