Preemptable Remote Execution Facilities for the V-System

Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton
Computer Science Department
Stanford University
Stanford, CA 94305

Abstract

A remote execution facility allows a user of a workstation-based distributed system to offload programs onto idle workstations, thereby providing the user with access to computational resources beyond that provided by his personal workstation. In this paper, we describe the design and performance of the remote execution facility in the V distributed system, as well as several implementation issues of interest. In particular, we focus on network transparency of the execution environment, preemption and migration of remotely executed programs, and avoidance of residual dependencies on the original host. We argue that preemptable remote execution allows idle workstations to be used as a "pool of processors" without interfering with use by their owners and without significant overhead for the normal execution of programs. In general, we conclude that the cost of providing preemption is modest compared to providing a similar amount of computation service by dedicated "computation engines".

1. Introduction

A distributed computer system consisting of a cluster of workstations and server machines represents a large amount of computational power, much of which is frequently idle. For example, our research system consists of about 25 workstations and server machines, providing a total of about 25 MIPS. With a personal workstation per project member, we observe over one third of our workstations idle, even at the busiest times of the day.

There are many circumstances in which the user can make use of this idle processing power. For example, a user may wish to compile a program and reformat the documentation after fixing a program error, while continuing to read mail. In general, a user may have batch jobs to run concurrently with, but unrelated to, some interactive activity. Although any one of these programs may perform satisfactorily in isolation on a workstation, forcing them to share a single workstation degrades interactive response and increases the running time of non-interactive programs.

Use of idle workstations as computation servers increases the processing power available to users and improves the utilization of the hardware base. However, this use must not compromise a workstation owner's claim to his machine: A user must be able to quickly reclaim his workstation to avoid interference with personal activities, implying removal of remotely executed programs within a few seconds time. In addition, use of workstations as computation servers should not require programs to be written with special provisions for executing remotely. That is, remote execution should be *preemptable* and *transparent*. By preemptable, we mean that a remotely executed program can be migrated elsewhere on demand.

In this paper, we describe the preemptable remote execution facilities of the V-system [4, 2] and examine several issues of interest. We argue that preemptable remote execution allows idle workstations to be used as a "pool of processors" without interfering with use by their owners and without significant overhead for the normal execution of programs. In general, we conclude that the cost of providing preemption is modest compared to providing a similar amount of computation service by dedicated "computation engines". Our facilities also support truly distributed programs in that a program may be decomposed into subprograms, each of which can be run on a separate host.

There are three basic issues we address in our design. First, programs should be provided with a network-transparent execution environment so that execution on a remote machine is the same execution on the local machine. By *execution environment*, we mean the names, operations and data with which the program can interact during execution. As an example of a problem that can arise here, programs that directly access hardware devices, such as a graphics frame buffer, may be inefficient if not impossible to execute remotely.

Second, migration of a program should result in minimal interference with the execution of the program and the rest of the system, even though migration requires atomic transfer of a copy of the program state to another host. Atomic transfer is required so that the rest of the system at no time detects there being other than one copy. However, suspending the execution of the migrating program or the interactions with the program for the entire time required for migration may cause interference with system execution for several seconds and may even result in failure of the program. Such long "freeze times" must be avoided.

Finally, a migrated program should not continue to depend on its previous host once it is executing on a new host, that is, it should have no *residual dependencies* on the previous host. For example, a program either should not create temporary files local to its current computation server or else those files should be migrated along with the program. Otherwise, the migrated program continues to impose a load on its previous host, thus diminishing some of the benefits of migrating the program. Also, a failure or reboot of the previous host causes the program to fail because of these inter-host dependencies.

The paper presents our design with particular focus on how we have addressed these problems. The next section describes the remote execution facility. Section 3 describes migration. Section 4 describes performance and experience to date with the use of these facilities. Section 5 compares this work to that in some other distributed systems. Finally, we close with conclusions and indications of problems for further study.

2. Remote Execution

A V program is executed on another machine at the command interpreter level by typing:

<program> <arguments> @ *<machine-name>*

Using the meta-machine name *:

<program> <arguments> @ *

executes the specified program at a random idle machine on the network. A standard library routine provides a similar facility that can be directly invoked by arbitrary programs. Any program can be executed remotely providing that it does not require low-level access to the hardware devices of the machine from which it originated. Hardware devices include disks, frame buffers, network interfaces, and serial lines.

A suite of programs and library functions are provided for querying and managing program execution on a particular workstation as well as all workstations in the system. Facilities for terminating, suspending and debugging programs work independent of whether the program is executing locally or remotely.

It is often feasible for a user to use his workstation simultaneously with its use as a computation server. Because of priority scheduling for locally invoked programs, a text-editting user need not notice the presence of background jobs providing they are not contending for memory with locally executing programs.

2.1. Implementation

The V-system consists of a distributed kernel and a distributed collection of server processes. A functionally identical copy of the kernel resides on each host and provides address spaces, processes that run within these address spaces, and network-transparent interprocess communication (IPC). Low-level process and memory management functions are provided by a *kernel server* executing inside the kernel. All other services provided by the system are implemented by processes running outside the kernel. In particular, there is a *program manager* on each workstation that provides program management for programs executing on that workstation.

V address spaces and their associated processes are grouped into *logical hosts*. A V process identifier is structured as a *(logical-host-id, local-index)* pair. In the extreme, each program can be run in its own logical host. There may be multiple logical hosts associated with a single workstation, however, a logical host is local to a single workstation.

Initiating local execution of a program involves sending a request to the local program manager to create a new address space and load a specified program image file into this address space. The program manager uses the kernel server to set up the address space and create an initial process that is awaiting reply from its creator. The program manager then turns over control of the newly created process to the requester by forwarding the newly created process to it. The requester initializes the new program space with program arguments, default I/O, and various "environment variables", including a name cache for commonly used global names. Finally, it starts the program in execution by replying to its initial process. The communication paths between programs and servers are illustrated in Figure 2-1.

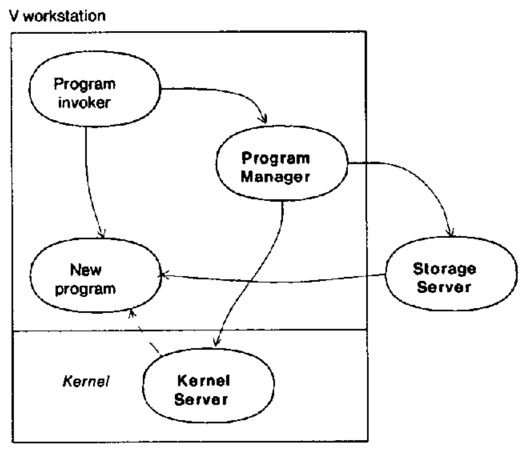

Figure 2-1: Communication paths for program creation.

A program is executed on another workstation by addressing the program creation request to the program manager on the other workstation. The appropriate program manager is selected using the process group mechanism in V, which allows a message to be sent to a group of processes rather than just individual processes [5]. Every program manager belongs to the well-known program manager group. When the user specifies a particular machine, a query is sent to this group with the specified host name, requesting that the associated program manager respond. The response indicates the program manager to which to send the program creation request. When the user specifies "*", a query is sent requesting a response from those hosts with a reasonable amount of processor and memory resources available for remotely executing programs. Typically, the client receives several responses to the request. Currently, it simply selects the program manager that responds first since that is generally the least loaded host. This simple mechanism provides a decentralized implementation of scheduling that performs well at minimal cost for reasonably small systems.

Beyond selection of a program manager, remote program execution appears the same as local program execution because programs are provided with a network-transparent execution environment, assuming they do not directly access hardware devices. In particular:

- The program address space is initialized the same as when the program is executed locally. For example, arguments and environment variables are passed in the same manner.
- All references by the program outside its address space are performed using network-transparent IPC primitives and globally unique identifiers, with the exceptions of the host-specific kernel server and program manager. For example, standard input and output are specified by global identifiers for the server processes implementing them.
- The kernel server and program manager on a remote workstation provide identical services for remotely executing programs as the local kernel server and program manager provide for locally executing programs.[1] Access to the kernel server and program manager of the workstation on which a program is running is obtained through *well-known local process groups*, which in this case contain only a single process. For example, the kernel server can be accessed by constructing a process-group-id consisting of the program's logical-host-id concatenated with the index value for the kernel server.[2] Thus, host-specific servers can be referenced in a location-independent manner.
- We assume that remotely executed programs do not directly access the device server[3] The limitation on device access is not a problem in V since most programs access physical devices through server processes that remain co-resident with the devices that they manage. In particular, programs perform all "terminal output" via a display server that remains co-resident with the frame buffer it manages [10, 14].

[1]It only makes sense to use the kernel server and program manager local to the executing program since their services are intrinsically bound to the machine on which the program is executing, namely management of processor and memory resources.

[2]A process-group-id is identical in format to a process-id.

[3]Actually, references to devices bind to devices on the workstation on which they execute, which is useful in many circumstances. However, we are not able to migrate these programs.

3. Migration

A program is migrated by invoking:

migrateprog [-n] [*<program>*]

to remove the specified program from the workstation. If no other host can be found for the program, the program is not removed unless the "-n" flag is present, in which case it is simply destroyed. If no program is specified, **migrateprog** removes all remotely executed programs.

A program may create sub-programs, all of which typically execute within a single logical host. Migration of a program is actually migration of the logical host containing the program. Thus, typically, all sub-programs of a program are migrated when the program is migrated. One exception is when a sub-program is executed remotely from its parent program.

3.1. Implementation

The simplest approach to migrating a logical host is to freeze its state while the migration is in progress. By *freezing* the state, we mean that execution of processes in the logical host is suspended and all external interactions with those processes are deferred.

The problem with this simple approach is that it may suspend the execution of the programs in the logical host and programs that are executing IPC operations on processes in the logical host for too long. In fact, various operations may abort because their timeout periods are exceeded. Although aborts can be prevented via "operation pending" packets, this effectively suspends the operation until the migration is complete. Suspension implies that operations that normally take a few milliseconds could take several seconds to complete. For example, the time to copy address spaces is roughly 3 seconds per megabyte in V using 10 Mb Ethernet. A 2 megabyte logical host state would therefore be frozen for over 6 seconds. Moreover, significant overhead may be incurred by retransmissions during an extended suspension period. For instance, V routinely transfers 32 kilobytes or more as a unit over the network.

We reduce the effect of these problems by copying the bulk of the logical host state before freezing it, thereby reducing the time during which it is frozen. We refer to this operation as *pre-copying*. Thus, the complete procedure to migrate a logical host is:

1. Locate another workstation (via the program manager group) that is willing and able to accommodate the logical host to be migrated.
2. Initialize the new host to accept the logical host.
3. Pre-copy the state of the logical host.
4. Freeze the logical host and complete the copy of its state.
5. Unfreeze the new copy, delete the old copy, and rebind references.

The first step of migration is accomplished by the same mechanisms employed when the program was executed remotely in the first place. These mechanisms were discussed in Section 2. The remainder of this section discusses the remaining steps.

3.1.1. Initialization on the New Host

Once a new host is located, it is initialized with descriptors for the new copy of the logical host. To allow it to be referenced before the transfer of control, the new copy is created with a different logical-host-id. The identifier is then changed to the original logical-host-id in a subsequent step (Section 3.1.3).

The technique of creating the new copy as a logical host with a different identifier allows both the old copy and the new copy to exist and be accessible at the same time. In particular, this allows the standard interprocess copy operations, ***CopyTo*** and ***CopyFrom***, to be used to copy the bulk of the program state.

3.1.2. Pre-copying the State

Once the new host is initialized, we pre-copy the state of the migrating logical host to the new logical host. Pre-copying is done as an initial copy of the complete address spaces followed by repeated copies of the pages modified during the previous copy until the number of modified pages is relatively small or until no significant reduction in the number of modified pages is achieved.[4] The remaining modified pages are recopied after the logical host is frozen.

The first copy operation moves most of the state and takes the longest time, therefore providing the longest time for modifications to the program state to occur. The second copy moves only that state modified during the first copy, therefore

[4]Modified pages are detected using dirty bits.

taking less time and presumably allowing fewer modifications to occur during its execution time. In a non-virtual memory system, a major benefit of this approach is moving the code and initialized data of a logical host, portions that are never modified, while the logical host continues to execute. For example, consider a logical host consisting of 1 megabyte of code, .25 megabytes of initialized (unmodified data) and .75 megabytes of "active" data. The first copy operation takes roughly 6 seconds. If, during those 6 seconds, .1 megabytes of memory were modified, the second copy operation should take roughly .3 seconds. If during those .3 seconds, .01 megabytes of memory were modified, so the third copy operation should take about 0.03 seconds. At this point, we might freeze the logical host state, completing the copy and transferring the logical host to the next machine. Thus, the logical host is frozen for about .03 seconds (assuming no packet loss), rather than about 6 seconds.

The pre-copy operation is executed at a higher priority than all other programs on the originating host to prevent these other programs from interfering with the progress of the pre-copy operation.

3.1.3. Completing the Copy

After the pre-copy, the logical host is frozen and the copy of its state is completed. Freezing the logical host state, even if for a relatively short time, requires some care. Although we can suspend execution of all processes within a logical host, we must still deal with external IPC interactions. In V, interprocess communication primitives can change the state of a process in three basic ways: by sending a request message, by sending a reply message, or by executing a kernel server or program manager operation on the process.[5] When a process or logical host is frozen, the kernel server and program manager defer handling requests that modify this logical host until it is unfrozen. When the logical host is unfrozen, the requests are forwarded to the new program manager or kernel server, assuming the logical host is successfully migrated at this point.

In the case of a request message, the message is queued for the recipient process. (The recipient is modified slightly to indicate that it is not prepared to immediately receive the message.) A "reply-pending" packet is sent to the sender on each retransmission, as is done in the normal case. When the logical host is deleted after the transfer of logical host has taken place, all queued messages are discarded and the remote senders are prompted to retransmit to the new host running these processes. For local senders, this entails restarting the send operation. The normal Send then maps to a remote Send operation to the new host, given that the recipient process is no longer recorded as local. For remote senders, the next retransmission (or at least a subsequent one) uses the new binding of logical host to host address that is broadcast when a logical host is migrated. Therefore, the retransmission delivers the message to the new copy of the logical host.

Reply messages are handled by discarding them and relying on the retransmission capabilities of the IPC facilities. A process on a frozen logical host that is awaiting reply continues to retransmit to its replier periodically, even if a reply has been received. This basically resets the replier's timeout for retaining the reply message so that the reply message is still be available once the migration is complete.

The last part of copying the original logical host's state consists of copying its state in the kernel server and program manager. Copying the kernel state consists of replacing the kernel state of the newly created logical host with that of the migrating one. This includes changing the logical-host-id of the new logical host to be the same as that of the original logical host.

Once this operation has succeeded, there exist two frozen identical copies of the logical host. The rest of the system cannot detect the existence of two copies because operations on both of them are suspended. The the kernel server on the original machine continues to respond with reply-pending packets to any processes sending to the logical host as well as retransmit Send requests, thereby preventing timeout.

If the copy operation fails due to lack of acknowledgement, we assume that the new host failed and that the logical host has not been transferred. The logical host is unfrozen to avoid timeouts, another host is selected for this logical host and the migration process is retried. Care must be taken in retrying this migration that we do not exceed the amount of time the user is willing to wait. In our current implementation, we simply give up if the first attempt at migration fails.

[5] We treat a *CopyTo* operation to a process as a request message.

3.1.4. Unfreezing the New Copy and Rebinding References

Once all state has been transferred, the new copy of the logical host is unfrozen and the old copy is deleted. References to the logical host are rebound as discussed next.

The only way to refer to a process in V is to use its globally unique process identifier. As defined in Section 2, a process identifier is bound to a logical host, which is in turn bound to a physical host via a cache of mappings in each kernel. Rebinding a logical host to a different physical host effectively rebinds the identifiers for all processes on that logical host. When a reference to a process fails to get a response after a small number of retransmissions, the cache entry for the associated logical host is invalidated and the reference is broadcast. A correct cache entry is derived from the response. The cache is also updated based on incoming requests. Thus, when a logical host is migrated, these mechanisms automatically update the logical host cache, thereby rebinding references to the associated process identifiers.

Various optimizations are possible, including broadcasting the new binding at the time the new copy is unfrozen.

3.2. Effect of Virtual Memory

Work is underway to provide demand paged virtual memory in V, such that workstations may page to network file servers. In this configuration, it suffices to flush modified virtual memory pages to the network file server rather than explicitly copy the address space of processes in the migrating logical host between workstations. Then, the new host can fault in the pages from the file server on demand. This is illustrated in Figure 3-1.

Just as with the pre-copy approach described above, one can repeatedly flush dirty pages out without suspending the processes until there are relatively few dirty pages and then suspend the logical host.

This approach to migration takes two network transfers instead of just one for pages that are dirty on the original host and then referenced on the new host. However, we expect this technique to allow us to move programs off of the original host faster, which is an important consideration. Also, the number of pages that require two copies should be small.

Finally, paging to a local disk might be handled by flushing the program state to disk, as above, and then doing a disk-to-disk transfer of program state over the network to the new host or the file server it uses for paging. The same techniques for minimizing freeze time appear to carry over to this case, although we can only speculate at this time.

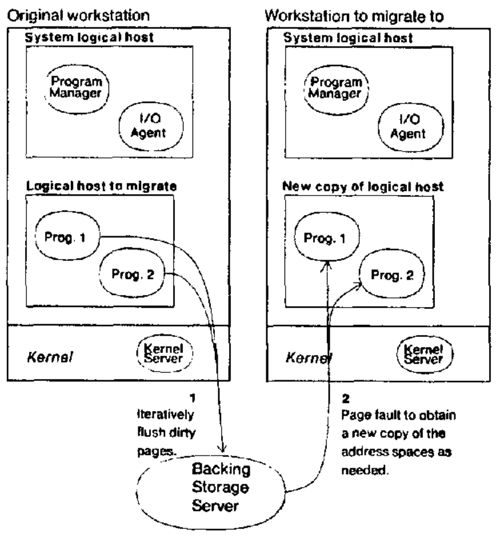

Figure 3-1: Migration with virtual memory.

3.3. Residual Host Dependencies

The design, as described, only deals with migrating programs whose state is contained in the kernel, the program manager and the address space(s) of the program being migrated. However, there are several situations where relevant state is stored elsewhere. This is not a problem if the state is stored in a globally accessible service, such as a network file server. However, extraneous state that is created in the original host workstation may lead to residual dependencies on this host after the program has been migrated. For example, the program may have accessed files on the original host workstation. After the program has been migrated, the program continues to have access to those files, by virtue of V's network-transparent IPC. However, this use imposes a continued load on the original host and results in failure of the program should the original host fail or be rebooted.

To remove this dependency it is necessary to identify and migrate (copies of) all files associated with the program being migrated. These files are effectively extensions of the program state, as considered in the above discussion, and could be moved in a similar fashion. We note, however, a few complications. First, a file may be arbitrarily large, introducing unbounded delay to the program migration. Second, a file that is being read may be a copy of some standard header file that also exists on the machine to which the process is being moved. Recognizing this case would save on file copying as well as reduce the final problem, namely, a program may have the symbolic name of the file stored internally, precluding changing the symbolic name when the file is migrated. However, this name may conflict with an existing file on the machine to which the file is moving. Related to this, a program may have written and closed a temporary file, yet be planning to read the file at some later point. There is no reliable way of detecting such files, so there is no reliable way to ensure they are migrated. With our current use of diskless workstations, file migration is not required and, moreover, file access cost is essentially the same for all workstations (and quite acceptable) [4, 11].

Similar problems can arise with other servers that store state specific to a particular program during its execution. Again, our approach has been to avoid local servers and thereby circumvent the problem of residual host dependencies.

4. Evaluation

At the time of writing, the remote execution facility has been in general use for one year and the migration facility is operational on an experimental basis. This section gives an indication of the cost of remote execution and migration, in both time and space, as well as general remarks about the usage we have observed.

4.1. Time for Remote Execution and Migration

The performance figures presented are for the SUN workstation with a 10 MHz 68010 processor and 2 Mbytes of local memory. Workstations are connected via a 10 Mbit Ethernet local area network.

The cost of remotely executing a program can be split into three parts: Selecting a host to use, setting up and later destroying a new execution environment, and actually loading the program file to run. The latter considerably dominates the first two. The cost of selecting a remote host has been measured to be 23 milliseconds, this being the time required to receive the first response from a multicast request for candidate hosts. Some additional overhead arises from receiving additional responses. (That is, the response time and the processing overhead are different.) The cost of setting up and later destroying a new execution environment on a specific remote host is 40 milliseconds. For diskless workstations, program files are loaded from network file servers so the cost of program loading is independent of whether a program is executed locally or remotely. This cost is a function of the size of the program to load, and is typically 330 milliseconds per 100 Kbytes of program.

The cost of migration is similar to that of remote execution: A host must be selected, a new copy of the logical host's state in the kernel server and program manager must be made, and the logical host's address spaces must be copied. The time required to create a copy of the logical host's kernel server and program manager state depends on the number of processes and address spaces in the logical host. 14 milliseconds plus an additional 9 milliseconds for each process and address space are required for this operation. The time required to copy 1 Mbyte of an address space between two physical hosts is 3 *seconds*.

Measurements for our C-compiler[6] and ***Tex*** text formatter programs indicated that usually 2 precopy iterations were useful (i.e. one initial copy of an address space and one copy of subsequently modified pages). The resulting amount of address space that must be copied, on average, while a program is frozen was between 0.5 and 70 Kbytes in size, implying program suspension times between 5 and 210 milliseconds (in addition to the time needed to copy the kernel server and program manager state). Table 4-1 shows the average rates at which dirty pages are generated by the programs measured.

Time interval (secs)	0.2	1	3
make	0.8	1.8	4.2
cc68	0.6	2.2	6.2
preprocessor	25.0	40.2	59.6
parser	50.0	76.8	109.4
optimizer	19.8	32.2	41.0
assembler	21.6	33.4	48.4
linking loader	25.0	39.2	37.8
tex	68.6	111.6	142.8

Table 4-1: Dirty page generation rates (in Kbytes).

[6]Which consists of 5 separate subprograms: a preprocessor, a parser front-end, an optimizer, an assembler, a linking loader, and a control program.

The execution time overhead of remote execution and migration facilities on the rest of the system is small:

- The overhead of identifying the team servers and kernel servers by local group identifiers adds about 100 microseconds to every kernel server or team server operation.
- The mechanism for binding logical hosts to network addresses predates the migration facility since it is necessary for mapping 32 bit process-ids to 48 bit physical Ethernet host addresses anyway. Thus, no extra time cost is incurred here for the ability to rebind logical hosts to physical hosts. The actual cost of rebinding a logical host to physical host is only incurred when a logical host is migrated.
- 13 microseconds is added to several kernel operations to test whether a process (as part of a logical host) is frozen.

4.2. Space Cost

There is no space cost in the kernel server and program manager attributable to remote execution since the kernel provides network-transparent operation and the program manager uses the kernel primitives. Migration, however, introduces a non-trivial amount of code. Several new kernel operations, an additional module in the program manager and the new command `migrateprog` had to be added to the standard system. These added 8 Kbytes to the code and data space of the kernel and 4 Kbytes to the permanently resident program manager.

4.3. Observations on Usage

When remote program execution initially became available, it was necessary to specify the exact machine on which to execute the program. In this form, there was limited use of the facility. Subsequently, we added the "@ *" facility, allowing the user to effectively specify "some other lightly loaded machine". With this change, the use increased significantly. Most of the use is for remotely executing compilations. However, there has also been considerable use for running simulations. In general, users tend to remotely execute non-interactive programs with non-trivial running times. Since most of our workstations are over 80% idle even during the peek usage hours of the day (the most common activity is editing files), almost all remote execution requests are honored. In fact, the only users that have complained about the remote execution facilities are those doing experiments in parallel distributed execution where the remotely executed programs want to commandeer 10 or more workstations at a time.

Very limited experience is available with preemption and migration at this time. The ability to preempt has to date proven most useful for allowing very long running simulation jobs to run on the idle workstations in the system and then migrate elsewhere when their users want to use them. There is also the potential of migrating "floating" server processes such as a transaction manager that are not tied to a particular hardware device. We expect that preemption will receive greater use as the amount of remote and distributed execution grows to match or exceed the available machine resources.

5. Related Work

Demos/MP provides preemption and migration for individual processes [15]. However, hosts are assumed to be connected by a reliable network, so Demos-MP does not deal with packet loss when pending messages are forwarded. Moreover, Demos/MP relies on a *forwarding address* remaining on the machine from which the process was migrated, in order to handle the update of outstanding references to the process. Without sophisticated error recovery procedures, this leads to failure when this machine is subsequently rebooted and an old reference is still outstanding. In contrast, our use of logical hosts allows a simple rebinding that works without forwarding addresses.

Locus provides preemptable remote execution as well [3]. However, it emphasizes load balancing across a cluster of multi-user machines rather than on sharing access to idle personal workstations. Perhaps as a consequence, there appear to have been no attempts to reduce the "freeze time" as discussed above. Moreover, in contrast to Locus, virtually all the mechanisms in V have been implemented outside the kernel, thus providing some insight into the requirements for a reduced kernel interface for this type of facility.

The Cambridge distributed system provides remote execution as a simple extension of the way in which processors are allocated to individual users [13]. Specifically, a user can allocate additional processors from the pool of processors from which his initial processor is allocated. Because no user owns a particular processor, there is less need for (and no provision for) preemption and migration, assuming the pool of processors is not exhausted. However, this processor allocation is feasible because the Cambridge user display is coupled to a processor by a byte stream providing conventional terminal service. In contrast, with high-performance bitmap displays, each workstation processor is closely bound to a particular display, for instance, when the frame buffer is part of its addressable memory. With this architecture, personal claim on a display implies claim on the attached processor. Personal claim on a processor that is currently in use for another user's programs requires that these programs are either destroyed or migrated to another processor.

Several problem-specific examples of preemptable remote execution also exist. Limited facilities for remote "down-loading", of ARPANET IMPs, for example, have been available for some time. Early experiments with preemptable execution facilities included the "Creeper" and "relocatable McRoss" programs at BBN and the "Worms" programs at Xerox [17]. Finally, a number of application-specific distributed systems have been built, which support preemptable remote executives (see, for example, [1, 9]). However, in each case, the facilities provided were suitable for only a very narrow range of applications (often only one). In contrast, the facilities presented here are invisible to applications and, therefore, available to all applications.

Our work is complementary to the extant literature on task decomposition, host selection, and load balancing issues (see, for example [6, 7, 8, 12]).

6. Concluding Remarks

We have described the design and performance of the remote execution facilities in the V distributed system. From our experience, transparent preemptable remote program execution is feasible to provide, allowing idle workstations to be used as a "pool of processors" without interfering with personal use by their owners. The facility has reduced our need for dedicated "computation engines" and higher-performance workstations, resulting in considerable economic and administrative savings. However, it does not preclude having such machines for floating point-intensive programs and other specialized computations. Such a dedicated server machine would simply appear as an additional free processor and would not, in general, require the migration mechanism.

The presentation has focused on three major issues, namely: network-transparent execution environment, minimizing interference due to migration and avoiding residual host dependencies. We review below the treatment of each of these issues.

The provision of a network-transparent execution environment was facilitated by several aspects of the V kernel. First, the network-transparency of the V IPC primitives and the use of global naming, both at the process identifier and symbolic name level, provide a means of communication between a program and the operating system (as well as other programs) that is network-transparent. In fact, even the V debugger can debug local and remote programs with no change using the conventional V IPC primitives for interaction with the process being debugged.

Second, a process is encapsulated in an address space so that it is restricted to only using the IPC primitives to access outside of its address space. For example, a process cannot directly examine kernel data structures but must send a message to the kernel to query, for example, its processor utilization, current-time and so on. This prevents uncontrolled sharing of memory, either between applications or between an application and the operating system. In contrast, a so-called *open system* [16] cannot prevent applications from circumventing proper interfaces, thereby rendering remote execution impossible, not to mention migration.

Finally, this encapsulation also provides protection between programs executing on behalf of different (remote) users on the same workstation, as well as protection of the operating system from remotely executed programs. In fact, the V kernel provides most of the facilities of a multi-user operating system kernel. Consequently, remotely executed programs can be prevented from accessing, for example, a local file system, from interfering with other programs and from crashing the local operating system. Without these protection facilities, each idle workstation could only serve one user at a time and might have to reboot after each such use to ensure it is starting in a clean state. In particular, a user returning to his workstation would be safest rebooting if the machine had been used by remote programs.

Thus, we conclude that many aspects of multi-user timesharing system design should be applied to the design of a workstation operating system if it is to support multi-user resource sharing of the nature provided by the V remote execution facility.

Interference with the execution of the migrating program and the system as a whole is minimized by our use of a technique we call *pre-copying*. With pre-copying, program execution and operations on this program by other processes are suspended only for the last portion of the copying of the program state, rather than for the entire copy time. In particular, critical system servers, such as file servers, are not subjected to inordinate delays when communicating with a migrating program.

The suspension of an operation depends on the reliable communication facilities provided by the kernel. Any packet arriving for the migrating program can be dropped with assurance that either the sender of the packet is storing a copy of the data and is prepared to retransmit or else receipt of the packet is not critical to the execution of the migrating program. Effectively, reliable communication provides time-limited storage of packets, thereby relaxing the real-time constraints of communication.

To avoid residual dependencies arising from migration, we espouse the principle that one should, to the degree possible, place the state of a program's execution environment either in its address space or in global servers. That way, the state is either migrated to the new host as part of copying the address space or else the state does not need to move. For example, name bindings in V are stored in a cache in the program's address space as well in global servers. Similarly, files are typically accessed from network file servers. The exceptions to this principle in V are state information stored in the kernel server and the program manager. These exceptions seem unavoidable in our design, but are handled easily by the migration mechanism.

Violating this principle in V does not prevent a program from migrating but may lead to continued dependence on the previous host after the program has migrated. While by convention we avoid such problems, there is currently no mechanism for detecting or handling these dependencies. Although this deficiency has not been a problem in practice, it is a possible area for future work.

There are several other issues that we have yet to address in our work. One issue is failure recovery. If the system provided full recovery from workstation crashes, a process could be migrated by simply destroying it. The recovery mechanism would presumably recreate it on another workstation, thereby effectively migrating the process. However, it appears to be quite expensive to provide application-independent checkpointing and restart facilities. Even application-specific checkpointing can introduce significant overhead if provided only for migration. Consequently, we are handling migration and recovery as separate facilities, although the two facilities might use some common operations in their implementations.

Second, we have not used the preemption facility to balance the load across multiple workstations. At the current level of workstation utilization and use of remote execution, load balancing has not been a problem. However, increasing use of *distributed execution*, in which one program executes subprograms in parallel on multiple host, may provide motivation to address this issue.

Finally, this remote execution facility is currently only available within a workstation cluster connected by one (logical) local network, as is the V-system in general. Efforts are currently under way to provide a version of the system that *can* run in an internet environment. We expect the internet version to present new issues of scale, protection, reliability and performance.

In conclusion, we view preemptable remote execution as an important facility for workstation-based distributed systems. With the increasing prevalence of powerful personal workstations in the computing environments of many organizations, idle workstations will continue to represent a large source of computing cycles. With system facilities such as we have described, a user has access to computational power far in excess of that provided by his personal workstation. And with this step, traditional timesharing systems lose much of the strength of one of their claimed advantages over workstation-based distributed systems, namely, resource sharing at the processor and memory level.

Acknowledgements

Michael Stumm implemented the decentralized scheduler and exercised many aspects of the remote execution early in its availability. Other members of the Stanford Distributed Systems Group have also contributed to the ideas and their implementation presented in this paper. Comments of the referees inspired a significant revision and improvement to the paper.

This research was supported by the Defense Advanced Research Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431.

References

1. J.-M. Ayache, J.-P. Courtiat, and M. Diaz. "REBUS, a fault-tolerate distributed system for industrial real-time control." *IEEE Transactions on Computers C-31*, 7 (July 1982), 637-647.

2. E.J. Berglund, K.P. Brooks, D.R. Cheriton, D.R. Kaelbling, K.A. Lantz, T.P. Mann, R.J. Nagler, W.I. Nowicki, M. M. Theimer, and W. Zwaenepoel. *V-System Reference Manual.* Distributed Systems Group, Department of Computer Science, Stanford University, 1983.

3. D.A. Butterfield and G.J. Popek. Network tasking in the Locus distributed UNIX system. Proc. Summer USENIX Conference, USENIX, June, 1984, pp. 62-71.

4. D.R. Cheriton. "The V Kernel: A software base for distributed systems." *IEEE Software 1*, 2 (April 1984), 19-42.

5. D.R. Cheriton and W. Zwaenepoel. "Distributed process groups in the V kernel." *ACM Transactions on Computer Systems 3*, 2 (May 1985), 77-107. Presented at the SIGCOMM '84 Symposium on Communications Architectures and Protocols, ACM, June 1984.

6. T.C.K. Chou and J.A. Abraham. "Load balancing in distributed systems." *IEEE Transactions on Software Engineering SE-8*, 4 (July 1982), 401-412.

7. D.H. Craft. Resource management in a decentralized system. Proc. 9th Symposium on Operating Systems Principles, ACM, October, 1983, pp. 11-19. Published as *Operating Systems Review* 17(5).

8. E.J. Gilbert. *Algorithm partitioning tools for a high-performance multiprocessor.* Ph.D. Th., Stanford University, 1983. Technical Report STAN-CS-83-946, Department of Computer Science.

9. H.D. Kirrmann and F. Kaufmann. "Poolpo: A pool of processors for process control applications." *IEEE Transactions on Computers C-33*, 10 (October 1984), 869-878.

10. K.A. Lantz and W.I. Nowicki. "Structured graphics for distributed systems." *ACM Transactions on Graphics 3*, 1 (January 1984), 23-51.

11. E.D. Lazowska, J. Zahorjan, D.R. Cheriton, and W. Zwaenepoel. File access performance of diskless workstations. Tech. Rept. STAN-CS-84-1010, Department of Computer Science, Stanford University, June, 1984.

12. R. Marcogliese and R. Novarese. Module and data allocation methods in distributed systems. Proc. 2nd International Conference on Distributed Computing Systems, INRIA/LRI, April, 1981, pp. 50-59.

13. R.M. Needham and A.J. Herbert. *The Cambridge Distributed Computing System.* Addison-Wesley, 1982.

14. W.I. Nowicki. *Partitioning of Function in a Distributed Graphics System.* Ph.D. Th., Stanford University, 1985.

15. M.L. Powell and B.P. Miller. Process migration in DEMOS/MP. Proc. 9th Symposium on Operating Systems Principles, ACM, October, 1983, pp. 110-119. Published as *Operating Systems Review* 17(5).

16. D.D. Redell, Y.K. Dalal, T.R. Horsley, H.C. Lauer, W.C. Lynch, P.R. McJones, H.G. Murray, and S.C. Purcell. "Pilot: An operating system for a personal computer." *Comm. ACM 23*, 2 (February 1980), 81-92. Presented at the 7th Symposium on Operating Systems Principles, ACM, December 1979.

17. J.F. Shoch and J.A. Hupp. "Worms." *Comm. ACM 25*, 3 (March 1982), .