Using Idle Workstations in a Shared Computing Environment

David A. Nichols
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

The Butler system is a set of programs running on Andrew workstations at CMU that give users access to idle workstations. Current Andrew users use the system over 300 times per day. This paper describes the implementation of the Butler system and tells of our experience in using it. In addition, it describes an application of the system known as *gypsy servers*, which allow network server programs to be run on idle workstations instead of using dedicated server machines.

1. Introduction

The Information Technology Center at Carnegie-Mellon University has spent the last four years developing the Andrew computing environment [5]. This environment consists of over 350 workstations running the 4.2 BSD release of the UNIX[1] operating system, and connected to a university-wide local area network. Andrew provides its users with a shared file system [6] and a network window manager that can display windows from programs running on other workstations. A number of these workstations are in faculty and staff offices, and the rest are in public terminal rooms available to the student body.

These workstations are meant for use by single users, and at any given time a large number of them are idle. This paper describes a system for making these idle workstations available to other users, called the *Butler* system, after Dannenberg's work [2]. The Butler system is a set of programs that run on Andrew workstations. It should run on any workstations that provide similar facilities, such as workstations running Sun's Network File System [9] and MIT's X window manager [7].

Other people have built systems similar to the Butler system. Shoch and Hupp's *worm* programs [8] used idle Alto workstations at Xerox PARC, but did not have access to a shared file system. Craft's resource manager [1] concentrated on building configurations of resources to provide various services. Hagmann [3], Litzkow [4], and Theimer [10] all provide systems quite similar to Butler. The main difference between these systems and the Butler system is that Butler can run on "off-the-shelf" operating systems without modification to the kernel or the application programs. However, this implementation choice limits the functionality that Butler can provide. One of the purposes of this paper is to show that such as system can be quite useful nonetheless, and to discuss some of the problems caused by these limitations.

The system maintains a list of the workstations that are in the pool of available machines. When a user wishes to use one of these workstations, he types

rem *command*

to the shell. The *rem* program finds one of the idle workstations and arranges to have the command executed on it. Every effort is made to make remote execution identical in effect to local execution. While this is not possible in every case, a good deal of compatibility can be achieved. Section 2.2 discusses this issue further.

These facilities are also available to C programmers via a subroutine library. The library is used by the *rem* program and by several experimental programs that use multiple machines.

The next section of this paper describes the implementation of this system within Andrew. Section 3 describes *gypsy servers*, which allow us to run network servers on idle workstations instead of using dedicated

[1] UNIX is a trademark of AT&T.

server machines. Section 4 tells of the usage the system has received to date. In section 5, we describe problems that arose during the building of the system, and how we did or did not deal with them. Finally, section 6 provides the conclusion to the paper.

2. Implementation

The Butler system is implemented as a set of programs running on the Andrew workstations. We made no changes to the kernel. Applications may run without change, but they can benefit from minor modifications.

2.1. Process Invocation

When a machine is donated to the pool of available machines, it runs a program called *butler* (sometimes referred to as "the butler"). This program is responsible for adding and removing the machine from the global free machine registry, and for handling requests for remote execution on that machine. The butler allows only one user to use the machine at a time, but it does allow that user to run as many processes as he wishes.

The process of claiming a machine and invoking a process on it is shown in figure 1. The *rem* program on the client machine first contacts the registry to find a candidate machine to use. The registry returns the name and net address of the potential server machine (this process is described in more detail below). The machine registry gives no guarantees about which machines are available, so the *rem* program contacts the candidate machine and checks to make sure that it is available.

If the machine is available, the butler marks it as in use and removes its name from the free machine registry. The *rem* program then sets up the remote execution environment, invokes the command, and waits for it to exit. The client is free to invoke several commands before freeing the machine, but the current *rem* program runs only one program before releasing the machine. When the client is done, it releases the machine, and its butler returns the machine to the free pool.

When the machine is reclaimed by its owner, usually by logging in at the console, the butler warns and kills any guest processes that may be running, removes the machine's name from the global registry, and exits.

2.2. The remote execution environment

When a program is running on a remote machine, *butler* and *rem* try to provide an execution environment that is as much like the user's execution environment as possible. Ideally, the program running on the remote machine would have access to the identical objects that it would have had access to on the user's machine. However, this is not always practical or desirable from a performance standpoint. For this reason, the kinds of resources a program uses fall into three groups.

Those provided by global servers. In Andrew, the main service provided by global servers is the file system. When a program is run on another machine, it simply contacts the same file servers that it would have contacted had it run on the local machine. Since the files are provided by servers in either case, there is no difference in semantics or performance when running a program remotely.

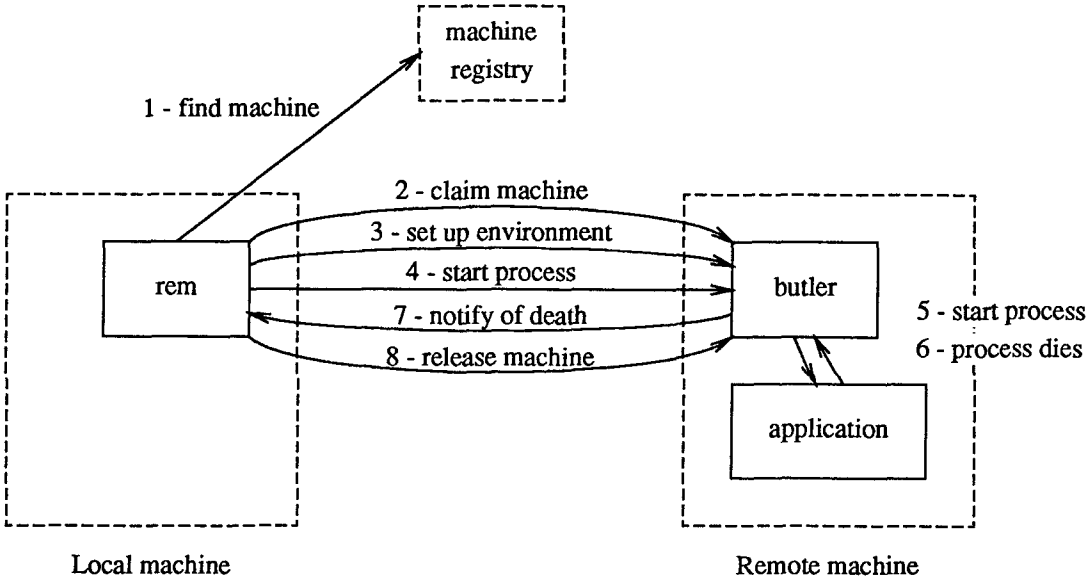


Figure 1: Process Invocation

The Andrew file system uses whole file transfer to move data to and from the server, and changes made to a file are not visible to other workstations until the file is closed. This prevents programs that do fine-grained sharing of file information from operating on different machines. However, programs that share at a file open/close granularity, such as a parallel *make* program, work fine.

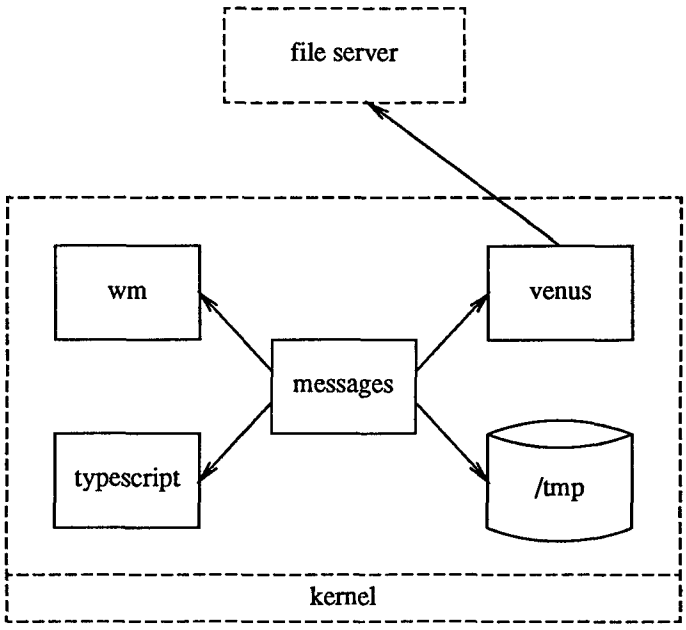
Those provided by the user's machine. In Andrew, applications communicate with the window manager via a network byte stream, even when the application is running on the same machine as the display driver. When the application is run on a remote machine, this network connection is established back to the user's window manager. It turns out that the performance of the window manager is nearly as good when the connection goes between machines as when it is local, so users do not see much of a performance penalty for running window manager applications remotely.

The standard input and output streams of a program are also directed back to the user's machine by use of a network byte stream. The *rem* program moves data between the end of the byte stream on the user's workstation and the user's terminal. Network byte streams and local terminals do not behave identically in UNIX, so the user sometimes sees minor differences in the way buffering of the standard input and output streams is handled in the remote case.

Those provided by the remote machine. Some resources are best handled by the remote kernel: examples are the time of day and software interrupts. Providing these resources on the remote machine instead of on the user's machine can cause problems, and the Butler system uses them only when it would be impractical to provide the original resource or when there would be a severe performance penalty for doing so. For example, using the time provided by remote workstation can cause incorrect timestamps to be written on files because the clocks are not synchronized. To alleviate this problem, Andrew workstations periodically resynchronize their clocks to the clocks of the file servers.

In Andrew, the "/tmp" directory used by many programs for temporary files is local to the workstation and is not kept in the shared file system. The result is substantial performance improvement since 300 workstations are not constantly trying to change the same directory. However, programs that run on one workstation cannot communicate with programs running on another by leaving files in the "/tmp" directory. We have not tried to split up applications that use "/tmp" for communication (such as the C compiler).

Figures 2 and 3 show the difference between the local and remote execution environments. In the figures, the *messages* program uses standard I/O for a greeting message, creates a window for mail reading, and uses the file system and kernel facilities. The *wm* program is the window manager and *venus* is the local cache manager for the Andrew file system. The figures show how access to the facilities used by *messages* is given in the local and remote cases.



Local workstation

Figure 2: The local execution environment

2.3. The machine registry

The butlers announce the presence of their machines in the free pool by writing a file to a special directory in the shared file system. This file contains the name and net address of the machine, along with any special attributes associated with the machine. For these attributes, the butlers list the CPU type and the version of the current software release for the machines. The list of available machines is only a hint for the *rem* program; it must still contact the butler on the machine that it chooses to ask if the machine indeed is available.

In early versions of the butler system, the *rem* programs scanned the list of butlers directly to find a suitable machine. However, this database is changing quite rapidly as butlers add and remove their entries, causing the search process to slow down with extra file fetches. To speed up the search process, a special server program, called *mreg*, runs on several machines and caches the information in these files. A simple RPC call to one these servers can quickly find a likely candidate for a free machine.

The *mreg* servers also register themselves in a global directory, just as the butlers do. But since this set of machines is changing far less rapidly than the one containing the list of butlers (once an hour vs. once a minute), it is usually cached on the user's workstation.

The *mreg* programs are started up on idle workstations by *butler* whenever it notices that too few are running. They are an example of *gypsy servers*, which are described in detail in the next section.

Now that we have used this system for a while, we believe that a better solution is to use the machine registry servers only, and to store no information about the available machines in the file system. Because all the free machines in the network are busy checking to make sure a registry server is running, the service is quite robust. Also, the butlers re-register themselves every 30 minutes in case the free machine database is damaged. With our current figure of 60 free machines on average, if a machine registry were to crash, another would be started and it would know about a few free machines within one or two minutes.

3. Gypsy servers

An interesting application of the Butler software is its use by the so-called *gypsy servers*. These are network servers that have no dedicated machines to run on. Instead, they float from workstation to workstation using the facilities of the butler to find free machines.

One such server is the help daemon. This program caches a list of all the help files in the system, keeping track of the help keywords for each file and the file's full pathname. When the help program wants to find the help file for some topic, it sends an RPC request to the help server and receives a list of files that contain help for the topic. Since the help files are scattered throughout many directories, using the help daemon is much faster than fetching each of the help directories from the file server and scanning them in the help program.

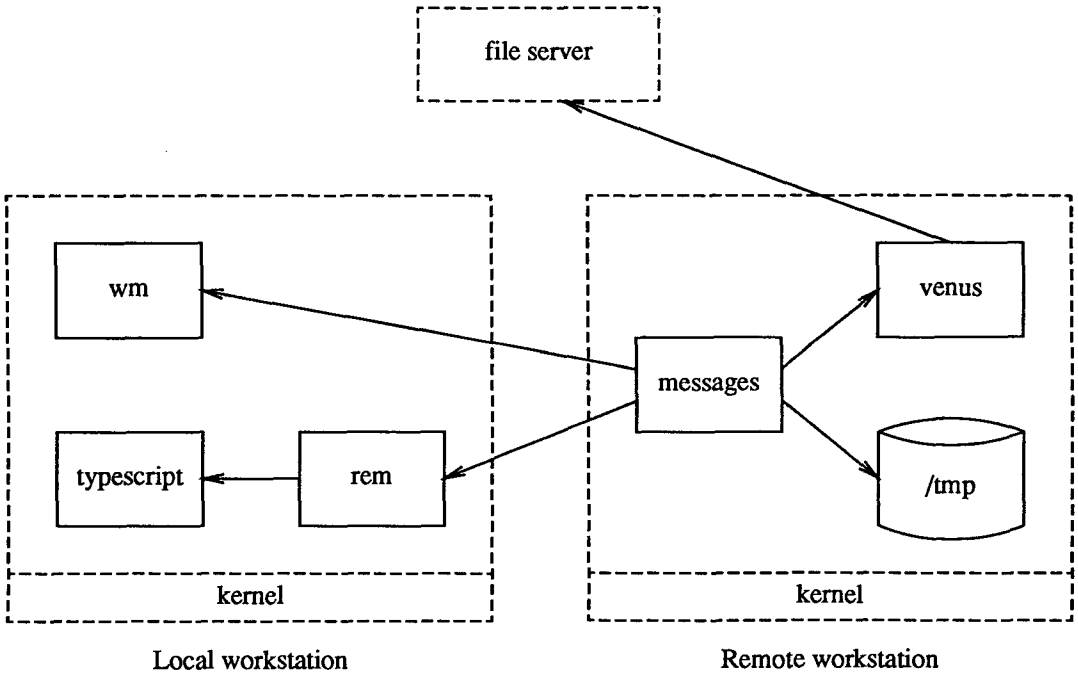


Figure 3: The remote execution environment

Another example are the machine registry servers described in the preceding section. The machine registry servers differ from other gypsy servers in that they are started by the butlers, while the other gypsy servers must be started in other ways.

3.1. How they work

The servers maintain a list of machines running the service by inserting and deleting files from a directory in the shared file system. Each file gives the name of the machine running the service and its network address. A client of the service contacts a server by reading the list of servers and choosing one to call.

Each server updates the modified time of its file every t seconds, where t is a constant of the particular service. Periodically, each server examines the list of running servers and verifies that all the files have a modified time that is within $2t$ seconds of the present. Those files that are too old are assumed to have been written by servers that have crashed, and are deleted. After doing the time check on each file, the server checks the number of servers that are running and verifies that it above the minimum number of servers for the program. If it is too small, another server is started using the *rem* program. The servers also check to ensure that some maximum number of servers is not exceeded.

The *mreg* servers used by the butlers are a special kind of gypsy service. Instead of being started by other *mreg* servers, they are started by the butlers running on idle workstations. This ensures that *mreg* servers are always running whenever any machines are in the free pool.

3.2. Problems with the servers

One problem with the gypsy servers is keeping them running at all times. Since the servers keep themselves running, it is possible that all the servers could die at once and nothing would be left to restart them. One can make this event less likely by increasing the number of servers running at all times. A better solution is to have a program on a dedicated machine that performs the periodic checks of the running servers described in the preceding section. Since this is a fairly inexpensive operation, this machine could also be used for other purposes, such as serving as someone's workstation.

It is also possible that the workstation pool could dry up, making the service unavailable for a while due to the lack of machines. To handle this case, the dedicated machine could take over the service temporarily until workstations became available again. For certain applications, the client programs can be designed to operate without the servers, doing the work locally instead. In either case, some sort of performance penalty is likely.

A second problem with the servers is keeping them authenticated with the Andrew file system at all times. The file system has an authentication scheme involving *authentication tokens*, which can be created when a user presents his password, and which give him access to the file system for up to 25 hours. These tokens are automatically copied to the remote machine by the *rem* program when it invokes a remote process.

The problem is that servers will have their tokens expire. Since they are unattended, there are no operators around to type in the password every day. Putting the password in the program itself isn't very secure and makes the password difficult to change. Putting it in a file in the shared file system causes startup problems since the password is unavailable at startup time. Again, the best solution involves using a dedicated machine to start the server programs. This machine would have the password on its local disk and could generate authentication tokens as needed. As before, the load on this machine would be light, so it could be used for other purposes or could be used to maintain several gypsy services.

4. Usage

The Andrew system currently has approximately 350 workstations on line, with about 3600 registered users. Of these workstations, about 50-70 workstations are in the free pool during the daytime. At night, the number of free machines goes up to over 100.

The number of invocations of the *rem* program has gone up from about 25 per day when the facility was introduced in September 1986 to about 300 per day in February 1987. Of these, about one-third are for the *typescript* program, which gives the user a shell on the remote machine. Each of the *typescript* invocations represents an extended session, so the actual number of commands actually executed on remote machines is much higher than 300 per day.

When we started this project, we envisioned that users would run mostly compilers, document processors, and other non-interactive, CPU-intensive applications. We were surprised by the large number of invocations of *typescript* and other interactive programs.

According to a survey of the butler users conducted as this paper was being written, the large proportion of *typescript* invocations is caused by several factors:

- The overhead of starting a remote program with *rem* is about seven seconds, while starting a command from a shell takes less than a second. By running *typescript*, users are effectively caching connections to other machines.
- If a user runs a typescript on the other machine, then he retains a handle on that

machine so that he can later find out what his programs are up to. The current system marks a machine as "in use" as soon as one command is run on it, so if a user invokes a command directly with the *rem* program, he cannot later run the process status command to find out what is going on if the program hangs. But if the user uses his one command to get a typescript, he can later use that typescript to examine the processes he has started on the remote machine.

- When a user uses the same machine for several commands, he can take advantage of the caching provided by the Andrew file system. If the user uses the *rem* program for each command, he receives a random sequence of machines and does not have any files cached between invocations.
- There are a few bugs in simulating a terminal for the standard I/O streams when commands are invoked directly from *rem*. By using a typescript, the user avoids these bugs.

As mentioned above, we were also surprised to learn that the majority of the uses of *rem* have been for interactive programs instead of computation-intensive programs. Even with the remote typescript invocations removed from the data, over half of the uses of *rem* were for interactive applications. In particular, the mail and bboard system accounted for 20% of the total invocations of *rem*. We believe that this is because the mail and bboard programs in Andrew are fairly memory intensive and users are taking advantage of the butler system to prevent their workstations from thrashing.

We are just beginning to develop parallel programs to increase the speed of applications directly, instead of just providing more cycles to people doing more than one thing at a time. For example, a parallel version of the UNIX *make* program is now working. In addition, Gregory McRae and Joseph Pekny of the Chemical Engineering Department are using groups of 50 or so machines to solve certain combinatorial search problems.

5. Practical considerations

As we were building and using this system, we encountered several problems. This section describes the problems, the approach we took to dealing with them, and the mixed degrees of success we had with the solutions.

5.1. Getting the machines

The original version of butler had to be run explicitly by the user in order to donate his machine to the pool. Users could set up command files to do this for them automatically whenever they logged in and out of their machines, but few users did this. As a result, we wrote *autobutler* and installed it on all workstations by administrative fiat. *Autobutler* runs all the time on the workstation and monitors the state of the machine to decide when it should be added to or removed from the free pool.

A user-created configuration file on the machine tells *autobutler* when the machine should be in the free pool. The configuration file consists of a boolean expression that can test the number of logged in users, the idle time of the console keyboard, and the time of day. The default expression is *users = 0*, which causes the machine to be in the free pool whenever no one is logged into the workstation. When a user logs into his workstation, it is automatically reclaimed and any guest processes are evicted. Our survey showed that users are happy with this default, and very few have changed it.

When we installed *autobutler*, we also installed a configuration file causing all workstations to be donated to the butler pool by default. Immediately, the number of free machines shot up and has stayed high ever since. The moral seems to be that while users will not go to any particular effort to have their machines donated to the butler service, neither will they go to any particular effort to prevent it, unless *butler* causes problems for them. So far, very few users have wanted to turn off butler service on their machine.

5.2. Machine types

The pool of workstations at CMU includes four kinds of machines: Sun 2 workstations, Sun 3 workstations, DEC MicroVaxes, and IBM RT PCs. When a user runs a compiler, he usually wants to compile his program for the same kind of machine as the one he is using. In any event, he wants control over which kind of machine is used.

The *rem* program uses the machine type attributes stored in the machine registry to pick machines of the proper CPU type for each invocation. A user can specify which machine type he wants, or say that any type will do. By default, he gets the same type as the one he is running on at the time.

This same mechanism is used to distinguish between machines running the two major software releases at CMU. Most machines run the production release, while a few (about 15%) run the newer, experimental version of the Andrew software. Again, users can ask for either kind of workstation or specify that either will do.

Files are named in the Andrew system so that most

commands work without regard to CPU type. For example, the file */usr/andrew/bin/messages* always refers to the mail reading program, regardless of CPU type or software release. Symbolic links local to the workstation cause this name, and those of other system files, to refer to the correct one of the eight possible versions.

The butler system lacks more advanced facilities for selecting workstations. For example, users cannot ask for "an RT or Sun 3, but not a Sun 2 or MicroVax." In addition, users have asked for facilities to define other attributes for workstations, such as the department that owns it, and whether it is in a public cluster or not.

5.3. Living without process migration

Some systems similar to the butler system (such as the V system [11]) provide *process migration* to move visiting processes off the workstation when the owner reclaims it. We judged that is was too difficult to provide process migration in our system, due to the extensive kernel modifications it would entail. Instead, we have chosen to kill visiting processes when a machine is reclaimed.

In order to warn both the user and the programs when a machine is being reclaimed, *butler* proceeds as follows:

1. It sends a message to the *console* program of the machine that initiated the guest processes. The *console* program is run by most Andrew users and is a machine monitor that displays various information such as the load of the workstation, file transfers in progress, the status of incoming mail, the time of day, etc. In particular, it displays messages generated by UNIX programs and can display messages sent to it from other workstations. The messages sent by *butler* tell the user that the machine he is using is about to be reclaimed.

2. Two minutes later, the butler delivers a software interrupt (called a *signal* in UNIX) to each of the guest processes on the machine. We intend that the processes will respond to this signal by saving any state that they have at the time. Most of the editors in common use in Andrew will checkpoint their buffers and exit on receipt of this signal. Programs that do not handle the signal die immediately.

3. After another 30 seconds, the butler kills any remaining processes with an uncatchable signal. All the remaining processes are terminated at this point.

Usually, this warning mechanism has proved sufficient for our use. Many programs such as compilations can simply be restarted when aborted by the butler. When the user is running an interactive program such as a mail reading program, the warning sent to the console gives the user time to save his state and to find another machine.

However, it is annoying to have to move to another machine, particularly if the user has the bad luck to choose a machine that is just about to be reclaimed. Our survey showed that most users thought the reclaim mechanism was reasonable, but that many of them were annoyed by having interactive programs seemingly killed at random. Many asked for some sort of process migration or a way to ask for a machine that would remain free for a given period of time. It would be nice to have some way of estimating how long the machine is likely to remain available, perhaps by asking the owner or basing the predictions on his past use. These methods would only be heuristics, but it should not be too hard to improve on the current totally random method of choosing machines that is used today.

5.4. Security

Dannenberg gives a description of the problems with protecting the users of a system such as this one from one another [2]. There are two basic problems: protecting the host computer from being disrupted by the guest programs, and protecting the guest programs from modified system software running on the host computer.

In the first case, we use the security provided the by the host operating system, in this case UNIX coupled with the Andrew file system. While this combination is not completely secure, it has usually provided enough protection for our purposes. We have only had one reported case of a guest user violating protection mechanisms and reading private files that the owner had stored on his local disk.

More common is accidental disruption caused by guest programs using up some workstation resource. Several times a guest program has filled up a disk partition by leaving many files in the "/tmp" directory. When the owner reclaims his machine, he can experience difficulties in running various programs until the "/tmp" is cleared out. *Butler* now removes the temporary files that the guest user leaves behind as part of its normal cleanup procedure. Another form of accidental disruption is the flushing of the file system cache contents caused by the guest's use of the machine. When the owner returns, he finds that his favorite files have been removed from the cache to make room for the files of the guest.

Protecting the guests from the programs on the host

computers is much more difficult. Since the owner of a workstation has complete control over the system software run on it, it is in principle easy for the owner to run a modified *butler* that disrupts the guest programs or uses the file system rights shipped along with the program to delete that user's files.

To our surprise and relief, attacks of this form have not yet been a problem with the current system, despite the fact that the user community for Andrew is quite large. If we do have such problems, we hope to be able to handle them with the logging provided by the system. Since users know which machines are being used to run their programs, the owners of that machine can be held accountable for disreputable actions performed by host software on that machine. For public workstations that have no owners, we hope to be able to impose restrictions to prevent users from installing system software.

6. Conclusion

We have been quite pleased with the reception that the butler system has had with the local user community over the past half year. During that time, it has become quite popular despite its shortcomings. In the course of supporting the system during this time, we were surprised by several things:

- Transparent access to the local window manager when running remotely is almost as important as transparent access to the file system. Far more interactive programs are run with our system than we anticipated.
- We thought that process migration would not be very important because we underestimated the amount of interactive use the system would get. While it is possible to live without process migration, it would make our system much more pleasant to use.
- Problems related to security and protection occurred much less frequently than we anticipated. Even with a large user community such as ours, users can be mostly cooperative.

We have only preliminary experience with gypsy servers, but we are quite encouraged by the results. We believe that they are a real alternative to dedicating large numbers of machines for certain network services. We are working on plans to use them for parts of the mail system, and for the workstation-based servers that IBM PC-class workstations need to attach to the Andrew file system.

7. Acknowledgements

Jim Morris and the referees gave me many valuable comments on earlier versions of this paper. I am also grateful for the assistance of Richard Cohn, Mike Kazar, and Sherri Menees.

References

1. Daniel H. Craft. Resource Management in a Decentralized System. Proceedings of the Ninth ACM Symposium on Operating Systems Principles, October, 1983, pp. 11-19.

2. Roger Berry Dannenberg. *Resource Sharing in a Network of Personal Computers.* Ph.D. Th., Carnegie-Mellon University, Dec. 1982.

3. Robert Hagmann. Process Server: Sharing Processing Power in a Workstation Environment. Proceedings of the Sixth International Conference on Distributed Computing Systems, 1986.

4. Michael J. Litzkow. Remote Unix: Turning Idle Workstations into Cycle Servers. Proceedings of the Summer 1987 Usenix Conference, June, 1987, pp. 381-384.

5. James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. "Andrew: A Distributed Personal Computing Environment". *Comm. ACM 29*, 3 (March 1986), 184-201.

6. M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The ITC Distributed File System: Principles and Design. Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Dec., 1985, pp. 35-50.

7. Robert W. Scheifler and Jim Gettys. "The X Window System". *ACM Transactions on Graphics 5*, 2 (April 1986), 79-109.

8. John F. Shoch and Jon A. Hupp. "The "Worm" programs—Early Experience with a Distributed Computation". *Comm. ACM 25*, 3 (March 1982).

9. Sun Microsystems, Inc. *Networking on the SUN Workstation.* 1986.

10. Marvin M. Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems.* Ph.D. Th., Stanford University, June 1986. Available as Stanford Computer Science tech. report STAN-CS-86-1128.

11. Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable Remote Execution Facilities for the V-System. Proceedings of the Tenth ACM Symposium on Operating Systems Principles, 1985, pp. 2-12.