01 Jan 1990

DAWGS - A Distributed Compute Server Utilizing Idle Workstations

Henry Clark

Bruce M. McMillin
Missouri University of Science and Technology, ff@mst.edu

Recommended Citation

H. Clark and B. M. McMillin, "DAWGS - A Distributed Compute Server Utilizing Idle Workstations," *Proceedings of the 5th Distributed Memory Computing Conference, DMCC 1990*, vol. 2, pp. 732 - 741, article no. 556276, Institute of Electrical and Electronics Engineers, Jan 1990.
The definitive version is available at https://doi.org/10.1109/DMCC.1990.556276

DAWGS - A Distributed Compute Server Utilizing Idle Workstations

Henry Clark and Bruce McMillin‡
Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401
email: henryc@cs.umr.edu

ABSTRACT

A collection of powerful workstations interconnected by a local area network can be utilized as compute servers when left idle by their owners. DAWGS allows users to submit jobs for execution on an idle workstation somewhere on a local area network. DAWGS uses a distributed scheduler and a bidding scheme to determine on which machine to run a process. DAWGS can properly redirect all the I/O of a remotely executing process and can checkpoint and then subsequently restart the process, even if the restart is on a different machine than the checkpoint. Our method is different from other work in that it is fault-tolerant and fully distributed.

1.0 Introduction

Within many organizations are powerful workstations which sit on a user's desk for their personal use. When the user goes home at night, the workstation sits unutilized and so all of the processing power of the CPU is wasted. Similarly, during large portions of the day, the workstation may sit idle, particularly while the user takes a meal break or is doing other things.

Many users have computing needs in excess of what their individual workstations can provide. However, one of the privileges of owning a workstation is that you have a given amount of computing power awaiting your use. The question then becomes one of attempting to allow users to access the vast amounts of computing power available on a collection of workstations on a local area network without adversely affecting the owner's use of the workstation. Additionally, the owner of a workstation should be able to specify whether he will or will not allow processes to be run remotely on his machine.

The Distributed Automated Workload balancinG System (DAWGS) is a system designed to allow users to tap into this vast processing power that is unutilized. DAWGS allows users to send programs to remote machines for execution and does this in a manner which is both transparent to the user and transparent to the process. DAWGS uses a distributed scheduler based on a bidding scheme [5] to determine on which machine to run a process. DAWGS is capable of checkpointing and subsequently restarting a process, even when the restart is on a different machine than the process was originally on.

DAWGS, unlike some systems previously written, does not require the user to perform any special actions other than tell DAWGS that the user wants to run a process remotely. In other words, the user need not link with any special libraries or call any special functions to run a process under DAWGS. The DAWGS daemon process and the special sections of code that are placed into the Unix kernel allow DAWGS to properly set up an execution environment for the process so that the process is unaware of the fact that it is running on a remote machine.

The system that DAWGS was implemented and tested on was a network consisting of a IBM PC/RT Model 135 file server running NFS and 10 IBM 6152's interconnected via a 4 megabit/second token-ring network. The computers ran IBM's Academic Operating System 4.3, a BSD 4.3 derivative. Each 6152 workstation has a small local disk used to hold swap/page areas and a small root and user partition.

2.0 Definitions

We define a *local area network* to be a network of homogeneous computers which aren't widely geographically distributed and that are interconnected by some networking mechanism such that message passing has a non-negligible delay. We define an *idle workstation* to be a workstation that has the following characteristics: 1) no user is running a process on the workstation, i.e. no users are logged in or running a process in the background. 2) if a user is running a process on the workstation, then the process must have been idle for more than β minutes (where β is a system parameter). Essentially, a workstation is idle if no one is using it, or if someone is using it, then he or she must have executed no commands for β minutes or have no processes running in the background. A workstation is *busy* if it is not *idle*, i.e. a user is logged on the workstation using it or has a process running on it

‡ This work was supported in part by the National Science Foundation under Grant Numbers MIP-8909749 and CDA-8820714, in part by the AMOCO Faculty Development Program, in part by the Manufacturing Research and Training Center (MRTC), and in part by the McDonnell Douglas Corporation.

0-8186-2113-3/90/0000/0732$01.00 © 1990 IEEE

in the background. We define a *distributed scheduler* as a mechanism whose components (dawgd) run on each workstation scheduling processes to be run by DAWGS. The DAWGS scheduler works independently of any other scheduler running on any other workstation on the local area network. We define the *remote machine* to be the machine to which a process is migrated for execution and the *source machine* to be the machine from which the user submits the process for execution. We define *transparent execution* to be execution that is location independent (from the process's point of view). The main thrust of transparent execution is that a process should be able to execute on any homogeneous machine as long as certain I/O functions are taken care of. We define the process of a user having *reclaimed* a workstation to have occurred when a user allows his workstation to become idle, i.e. he doesn't use it for some period of time, and then the user resumes using the workstation again. We define an *interactive* process to be one with which the user must interact with from time to time during the process's execution, which is contrasted to a *batch* process which requires no interaction with the user from the time it starts execution until it completes execution.

3.0 Previous Work

Prior work in this area has yielded a variety of results. In the Butler System[1], Nichols implemented a system which allowed users to execute jobs, including interactive shells, on remote workstations, but did not allow process migration or the automatic use of logged, but idle, workstations. In Litzkow's work [10], a system called Remote Unix allow users to submit long-running jobs for remote execution, but required the user to link with a special library and had a centralized scheduling and allocation facility. De Witt, Finkel, and Soloman [8] describe the Crystal system in which processes may be run on different computers using a variety of mechanisms including a remote unix facility, but they used a centralized scheduler and the remotely-running process couldn't spawn additional child processes. The DEMOS/MP system [7] as described by Powell and Miller allowed processes to migrate in a distributed environment without using a centralized controller but used a custom operating system within which they operated. The V-System [9] described by Theimer, Lantz, and Sheriton used a remote execution facility on diskless workstations to execute processes on remote idle workstations, but didn't try to utilize idle logged stations. In the LOCUS distributed system [6], Walker, Popek, English, Kline, and Thiel show a distributed operating system upwardly compatible with Unix allowing remote process execution, but they also used a custom distributed operating system in which to operate. Baumgartner et. al. [11] in GAMMON show an implementation which uses load balancing across a network, but does not try to find the minimally loaded host or allow migrations subsequent to the initial job migration. Litzkow, Livny, and Mutka [2] developed a system for the execution of long-running jobs called Condor which used a centralized scheduler, didn't allow users to run jobs interactively, and required users to link with a special library. In the DUNIX operating system, Litman [17] shows an implementation of a non load-balancing Unix distributed over several VAX computers. Process Server [3] by Hagmann used a centralized load scheduler to assign work to a specialized series of processors which had characteristics such as bigger memory or a faster processor. The Amoeba Operating System [26] by Renesse, Staveren, and Tanenbaum is a custom-written object-oriented distributed operating system which allows users to allocate processors from a pool for their needs. Wills [22] presents a service execution mechanism for a distributed computing environment which allows users to access services offered by heterogeneous processors in a transparent manner, but really didn't offer any type of load balancing. Tanenbaum and Van Renesse [15] show additional issues in the formulation of distributed computing systems.

Other work has been done in related areas. Ni, Xu, and Gendreau [4] present a drafting algorithm for dynamic process migration in which processes use a drafting algorithm to choose a processor to execute a process on. Stankovic and Sidhu [5] present an adaptive bidding algorithm for finding the ideal candidate to migrate a job to. Bryant and Finkel [16] present a stable distributed scheduling algorithm in which processors cooperated in making scheduling decisions without a centralized controller. Efe and Groselj [24] present a load balancing algorithm which attempts to work optimally under very heavy loading and additionally attempts to minimize the control overhead and messages needed. Mirchandey, Towsley, and Stankovic [25] present a study of the performance characteristics of load-sharing algorithms. Chang and Maxemchuk [18] describe a reliable message broadcast protocol such that all receivers in a group receive the messages in the same order as they were transmitted. Powell and Prescotte [19] show additional work in the area of reliable broadcast communication mechanisms. Kleinrock and Korfhage [23] analyze the method of using idle workstations as compute servers and conclude that the use of many powerful idle workstations can have the same kinds of throughput as a large computing machine. Zayas [20] presents a copy-on-reference process migration mechanism where the real memory pages aren't copied until referenced, thus speeding up process migrations. Chou and Abraham [21] describe a method to assign modules of a process to heterogeneous processors such that the assignment is optimal. Theimer and Lantz [12] show that a simple distributed scheduler is optimal and show that such schedulers can be fault tolerant. Kruger and Livny [13] show that the addition of a process migration mechanism to a remote execution mechanism which allowed migrations after the initial migration was desirable.

4.0 An Overview of DAWGS

4.1 An Overview

Within DAWGS are three separate processes - the front-end process, the daemon process, and the i/o server process. The front-end process is the user interface to the DAWGS program - users invoke this front-end program to submit programs for remote execution. The daemon process (dawgd) is the program which records all load information, does all necessary bidding, checkpoints and initiates the restart of migrated jobs, and does some important bookkeeping. The i/o server process is responsible for redirecting all output of interactive processes back to the machine the user is sitting in front of.

An example of how a user interacts with DAWGS is pictured in the Figure 1.

```
mcs213k> cc -o testprog testprog.c
mcs213k> dawgs -s testprog
DAWGS: Submitting Job....
mcs213k>

...

You have new mail.
mcs213k> mail
Mail version 5.2 6/21/85. Type ? for help.
"/usr/spool/mail/henryc": 1 message 1 new
N 1 henryc  Thu Mar 29 17:30  11/280  "Your DAWG
& q
mcs213k>
```

Figure 1: DAWGS Usage Example

To use DAWGS, a user will submit a process for remote execution by using the *dawgs* command. He can control whether the job is an interactive or batch process by specifying different command-line options. If the process is an interactive process, DAWGS will start an I/O server to interact with the user while the process is running, otherwise all output will be returned to the user at the process's termination and no input (other than that from data files) will be supplied to the process. DAWGS will then move the process to a remote machine for execution and then move the results back to the source machine. DAWGS will also notify the user by mail that the process has completed.

Another major part of DAWGS is the kernel-resident code (kernel hooks) which alerts the daemon process of the activities of remotely running processes. Also contained within the kernel is the code which does the actual restart of a previously checkpointed process.

4.2 The Front-End Process

The front-end process serves several important functions. First, it provides a separate and clean user interface to the daemon process. Second, it does preliminary error-checking to ensure that, for example, the program the user wants to execute remotely exists and can be executed. Third, it passes important user-environment information to the daemon process such as the user's terminal type and the user's current window size, the user's uid, and the value of all currently-defined environment variables. Finally, it provides a convenient mechanism for starting the daemon, stopping the daemon, and forcing the daemon to print all accumulated statistics.

4.3 The Daemon Process (dawgd)

The daemon process (dawgd) is the heart of DAWGS. It is responsible for all tasks, except for those tasks which have been specifically given to another process (such as the i/o server or the front-end process). It is, for example, responsible for determining if a machine is busy or idle and detecting when a transition from busy to idle or idle to busy has occurred, forcing a checkpoint of a process, determining if a process needs to be migrated as a result of a idle to busy transition, and handling all incoming messages and routing them to the proper function within the daemon.

All information that the daemon process needs while a process is remotely running is kept in a special structure inside of the daemon called a queue structure, which is described in Figure 2.

child
current_chkpt
cwd[PATHNAMELEN]
env
last_ckpt
interactive_flag
job_name[PATHNAMELEN]
job_number
num_open_files
open_filename[MAXNUMFILES]
orig_filename[MAXNUMFILES]
pid
stat
remote_machine[MAXHOSTNAMELEN]
return_flag
source_machine[MAXHOSTNAMELEN]
term
uid

Figure 2: Queue Structure

The queue structure contains the source machine name, the remote machine name, the pid and uid of the process, the number of files that the process has open and their filenames, and other relevant environment information. This structure is created when the user submits a process for execution by DAWGS and follows the process as it is moved amongst the machines during its execution.

4.4 The I/O Server

The I/O server process is responsible for redirecting the output (standard input, standard output, and standard error) of all processes running interactively under DAWGS. It is not responsible for handling any other type of data input or output to the process through data files, sockets, and the like. A special server is started for interactive processes because we don't want the user to notice a degraded response time in handling the keystrokes they would type or in the display of any output of a remotely-running process (which could happen if we forced the daemon to handle this additional task).

5.0 Detailed Description of DAWGS

5.1 Remote Process Execution

Figure 3 describes the path a process follows while within DAWGS.

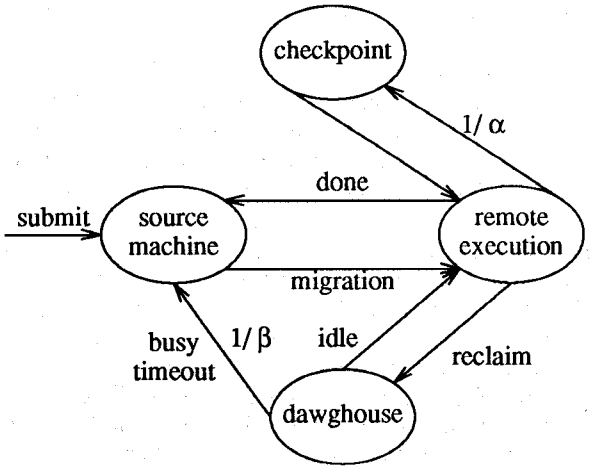

Figure 3: Process Path within DAWGS

When a process is submitted to the source machine, it seeks bids to find a remote machine on which to run the process. Upon finding a successful bidder, the source machine migrates the job and all other associated files to the remote machine for execution. When the process completes on the remote machine, the results are returned back to the source machine and the user is notified that his process has run to completion. If the remote machine makes an idle-to-busy transition while a remote process is running on it (i.e. a user *reclaims* the workstation or a new user logs onto the workstation), the remote process is stopped and placed in the dawghouse at which time a timer begins. If the machine becomes idle before the timer expires, then the process is restarted. The dawghouse is the state where a process is kept if it's stopped because of an idle-to-busy transition of the remote machine its running upon. If the timer expires and the machine is still busy, the process is migrated back to the source machine which will then find a new remote machine on which to run the process.

5.2 Finding a Remote Machine

The DAWGS daemon process keeps a load table which lists each machine that the daemon process is running on and the currently perceived load of that machine. The daemon determines which machines it can run on by reading a file at boot-time and can be forced to reread that file at any time by sending the daemon a signal. The load of a machine is described as either idle or busy, depending on whether a user is using the computer or not. When the daemon process on the source machine wishes to find a machine to execute a process upon, it searches the load table looking for machines it thinks are idle. Upon finding one or more idle machines, it sends a request for bids to the remote machine.

The bid that the remote machine returns is essentially the time that the remote machine has been idle (measured in seconds). If a remote machine has transitioned to busy, it adjusts the bid adversely such that its bid is guaranteed not to be accepted. If all bids returned to the source machine are bad, then the source machine sends a forced request for bids message to all machines running the DAWGS daemon. If all bids are still returned bad, then the daemon merely waits for δ minutes and then retries the bidding process. Note that this process accounts for inconsistencies in the load table which may occur because of errors introduced into the load table by a machine improperly reporting itself being busy or idle and for broadcast messages which might be missed by a daemon on any machine running dawgd.

When a machine transitions from a busy state to an idle state or from an idle to a busy state, the daemon process broadcasts a message to all other daemons on all other machines notifying them of the state change.

5.3 Checkpoint/Restart

Every α minutes, the daemon process on the remote machine stops the process that's running remotely and checkpoints the process. At the same time, it checkpoints any open data files so that if the process is restarted later it can resume processing any data files.

To checkpoint a process, the daemon process starts by creating a checkpoint file and writing the queue structure for the process. It then writes the Unix user (u) structure [14] for the process. The u structure contains all the information that the system will need in order to restart the process such as the size of the page table, the process control block, structures that hold information about open files, and other system information. The daemon then writes the proc structure for the process to disk. The proc structure contains all the process-related information such as the process pid and the like.

After having written all the control structures to disk, DAWGS then goes through all of the process's data and stack page tables and writes all of those pages to disk. DAWGS doesn't write the text pages to disk since they

can be retrieved from the original executable should they be needed in a restart of the process after a migration. Additionally, DAWGS makes its own copy of the original executable so that the original text segment may be retrieved by DAWGS at will.

Note that the checkpoint file(s) are kept on the disk local to the remote machine. Since one of the design objectives of DAWGS was to minimize the interference of DAWGS with any machine or with the network, we decided to keep the files locally and move them only when some extraordinary event happened (such as the forced migration of the process or the completion of the process). One might wish to only keep a current executable on the remote machine and store all other files on the source machine, but this will substantially increase the amount of network overheard imposed by DAWGS since all checkpoints and I/O performed by a remotely executing process, are done over the network, instead of just the local machine.

To restart a process, DAWGS performs the equivalent of a Unix *execl* and the process is illustrated in Figure 4.

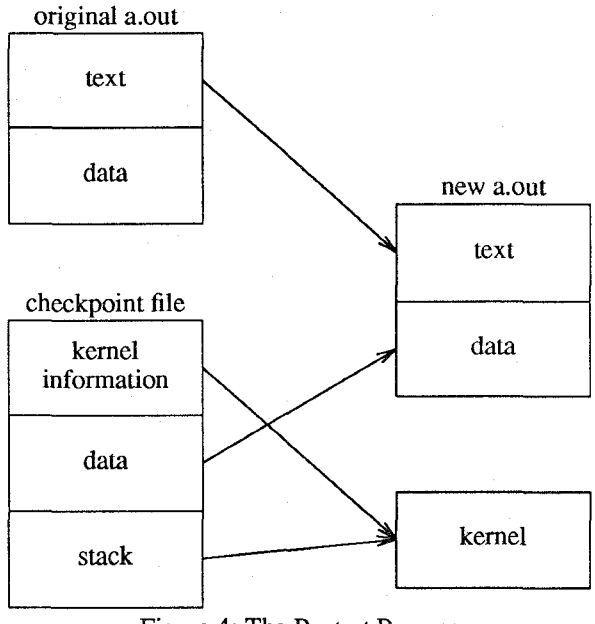

Figure 4: The Restart Process

Before the daemon process calls the kernel restart routine, the process must be restored to an a.out form so that the system can page from the disk image instead of having to preload the process. This is an advantage, particularly when faced with the prospect of loading a large image into an otherwise memory-limited workstation. The daemon reads the checkpointed data segment image and integrates it with the text segment and other information in the original a.out file and makes a new a.out file.

After the kernel restart routine is called, it first retrieves the Unix u structure and proc structure and restores relevant information from them into the process's u structure and proc structure. The restart routine then releases all the resources associated with the forked process and proceeds to customize the resources that the new process will need. It first allocates page tables for the new process and then forces the system to map the pages to the new a.out image. It then allocates stack space and reads the contents of the stack segment from the checkpoint file into the stack segment of the process. Finally, it restores all the process control block values and performs a final check to insure that all is restored properly. It then starts the process to running.

Note that before the process is restarted by the "execl" process just described, the daemon restores all file descriptors to their previous states. During the migration, the data files were placed in the DAWGS temporary directory, so it appears to the remotely running process that they are in the same place as they always have been. If data files are open, their file pointers are repositioned to where they were when the process was checkpointed.

5.4 I/O Redirection

If a process opens a disk file during the course of its execution, DAWGS must intercept the open because the file cannot be assumed to reside in the same place on the remote machine as it did on the source machine or there may be differing copies of a file with the same name on different machines (the password file is one example). To achieve this transparency, DAWGS intercepts the open call that the process makes and copies the data file that the process wants to access from the source machine to the remote machine. On the remote machine DAWGS places the file under a special temporary directory and then tells the kernel to open that file instead of the file that the program wanted to open. The kernel then proceeds to open the file and the program is unaware that DAWGS has redirected the open call.

When the process is done with the data file or has terminated, DAWGS retrieves the original filename that the file was stored under on the source machine and then copies the file back to the source machine. Its important that DAWGS monitor the close() function since the process may close the data file and then reopen it later, expecting the data file to be changed from earlier in the execution.

Sockets are currently unsupported in DAWGS. Plans are underway to incorporate support for them via the I/O server process so that a process may attach to a socket controlled by an I/O server and never know that it has been moved around by a migration.

5.5 Kernel Hooks

The kernel hooks are used to inform the DAWGS daemon of activity that the remotely-running process is attempting to perform. For example, a kernel hook is used to notify the kernel when a remote process attempts to open a file

or close a file. Generally speaking, all system calls that enter the kernel and return some type of machine-dependent result must have a kernel hook. Examples of these include creat(), open(), close(), gethostname(), fork(), getdomainname(), mkdir(), and the like.

A typical kernel hook resides in each appropriate system call and looks at the process group that the calling process belongs to. If the calling process is in a special process group to which only the daemon process and the remote process belong to, then the pid of the process is examined to determine if the process is the daemon or the remote process. If the process is the remote process, a signal is sent to the daemon and the daemon then takes the proper action (redirecting the open of a file, for example, or some other appropriate action) and then the remote process resumes execution.

Its important that the daemon set the process group properly when the remote process is first started for this checking to work properly. The process group is made unique in order to reduce the time needed to determine if the DAWGS daemon should be notified of an event. This is particularly important since the kernel hooks must be in some heavily utilized kernel code.

5.6 Process Migration

Process migration takes one of two forms - migration from the source machine to the remote machine and migration from the remote machine back to the source machine. Migration from the source machine to the remote machine takes place when a machine is picked by the bidding process to run the process and the process is then migrated. Migration from the remote machine back to the source machine occurs under two circumstances: 1) when the remote machine transitions from the idle state to the busy state and remains busy for β minutes, then the daemon on the remote machine will migrate the process back to the source machine for subsequent migration to a new remote machine. and 2) when a machine crashes and restarts and is busy upon restart, DAWGS can detect that it was running a process and will attempt to migrate the process back to the source machine for migration to a new remote machine.

In either case of forced migration, the logged-in user of the workstation will be impacted for just a few seconds while DAWGS migrates the files associated with the process back to the source machine. While this violates our goals of making DAWGS run transparently on a machine, it reduces the total load we place on the network when considering the network as a whole. Processes are actually migrated more than is minimally needed, particularly when a idle to busy transition is made on the remote machine. However, the migration back to the source machine must be made, particularly with interactive processes, since all the appropriate data and i/o server connections must be made from the source machine to the remote machine.

Interactive processes which are running remotely are handled slightly differently in the event that they need to migrate because a workstation has transitioned from an idle state to a busy state. When a workstation transitions and the daemon stops the remotely-running process, the daemon examines a flag in the queue structure to see if the process is running interactively. If the process is interactive, the daemon immediately seeks bids for another machine, and if the bidding is successful the process is immediately migrated and restarted, with little interference or noticeable disruption to the user. If the bidding process is unsuccessful, the user is given the choice of killing the process or waiting until a machine becomes available.

5.7 Fault Tolerance Features of DAWGS

DAWGS is fault-tolerant to most common user errors and to most system errors. Once a job is submitted to DAWGS and migrated to the remote machine, the process will eventually be run to completion and the results returned to the user. We don't, however, guarantee any type of response time or guarantee a minimum response time - we only guarantee that the job will be run at some point in the future.

Once the process has been migrated to the remote machine, the remote daemon checkpoints the process every α minutes to ensure that no significant amounts of CPU processing is lost. If the daemon or computer crashes before the job is done, that fact will be detected by the daemon on the remote machine the next time the daemon starts and actions will be taken to reload the dead process and run it to completion or to migrate it to the source machine so that it may be restarted. We assume that all checkpoints are contained on stable storage.

An additional fault-tolerant feature of DAWGS is that it periodically checks to ensure that its peers are still functional. Once every ε seconds, a dawgd will send a message to all of the other daemon processes on the network. If it doesn't get a response from each of the other daemons within a specific time period then the local daemon assumes that the remote daemon has crashed and is no longer functional. Once it assumes this, it updates its internal tables to reflect that fact. Once it has done this, though, it checks all messages it can listen to to see if in fact it has assumed this in error, and if it hasn't, then it marks the remote daemon as being active again.

5.8 Security Features of DAWGS

Many interesting security problems arise when one attempts to run a process remotely. The problems generally group themselves into the problem of the remote process disrupting the remote workstation or even causing the remote workstation to crash. DAWGS doesn't attempt to sidestep the issue of security, rather, we let the operating system worry about it by setting up an execution environment similar to one the user would see if he ran his process locally. We go to great trouble to set all

environment variables the same as they would be on the remote machine. Just in case the user would happen to change them, we try and determine if they are set to reasonable values before we allow the process to execute remotely. We also set the internal kernel structures to be the same as they would be on the source machine (well, as much as is possible). If a user can't access a file from the source machine, he won't be able to on the remote machine either. If the process will cause the source machine to crash, it will cause the remote machine to crash as well. Our approach is that the user should have the *same* privileges on the remote machine as he does on the local machine. We also don't allow system privileged accounts such as root to use DAWGS so that user's can't exploit any security holes.

6.0 Results

6.1 Fixed Settings

We set the value of α (the time between checkpoints of a remotely running process) to 30 minutes. In its basic form, α is just a measurement of the amount of CPU time that you want to waste in a worst case scenario (that being running until just shy of α and then crashing). Measurements of the reboot times of the machines that we ran dawgd on showed that the machines crashed and rebooted once every 2.7 days. The α we chose seems to be a good compromise between the goal of minimizing the amount of CPU time wasted and the amount of disk space containing previous checkpoints. The value of α is also affected by the value you choose for β, since you could lose α-1 time units of CPU processing when an idle-to-busy transition is made and the process has to be migrated to another machine. We chose to retain the most recent two checkpoints when a long-running process makes multiple checkpoints.

We set the value of β (the amount of time between an idle-to-busy transition and when we finish waiting and migrate the process back to the source machine) to 20 minutes. While this may seem high, you have to account for two different types of users. The first type of user will logon the workstation and stay active for a very long time. For this type of user, we want to move the process back for migration to a new machine as quickly as possible. The second type of user will only logon the workstation for a few minutes and then logoff. For this type of user we want to merely stop the process by putting it in the dawghouse while he uses the workstation for a short period of time and then restart the process after he leaves. We observed that 48.6% of our users used the computer for 20 minutes or less while those that remained for more than 20 minutes tended to stay active for a very long time and we define that length of time that the user stayed active as T. The value of β we chose allowed most short-term users to use the machine without forcing a migration of a remote-running process but moved the process when a long-term user logged onto the workstation. The important point is that we don't want to needlessly migrate a job, since this is an expensive operation.

We set the value of δ (the time between the failure of a forced bid and the time we start the bidding process again) to 10 minutes. Recall that a forced bid occurs when the daemon on the source machine has broadcast for bids and all bids that were returned were *bad*, that is, all the remote machines didn't want a remote process (usually because they already were running a process remotely or were busy because a user was logged onto the workstation). This time needs to be set somewhere in an intermediate range where we won't try too soon again and fail the bidding process again and a point where we don't waste too much time after a workstation has made a busy-to-idle transition and we can once again use the workstation. This particular value also allows enough time for a workstation to crash and reboot, just in case we would be able to use it when it rebooted and the daemon on that machine resumed execution.

We set the value of ε (the time between broadcasts to determine the state of other daemons on the network) to 10 minutes. This we felt would be reactive enough to machine crashes yet would allow machines to crash and restart transparently to the daemon on the remote machine (since a machine could crash and reboot within this time).

6.2 Measured Results

In attempting to ascertain whether DAWGS would provide a performance improvement, we first had to determine how much time each of our workstations spent in the *idle* state. Recall that a workstation is *idle* if no users are using it, or if a user is using it, then he/she must have not executed any commands for β minutes. We measured the utilization of our workstations over a one-month period and determined that our workstations were idle 60% of the time. Our daemon consumed less than 1% of the available CPU cycles on the workstation, thus not impacting the load noticeably upon the workstation.

Bidding is one of the most important functions that the daemon process performs. We measured the time from the original bid request broadcast to the final selection of a bid as taking, on average, 5.25 seconds and is referred to as B_t. This intermediate result for bidding is indicative of two different bid scenarios. The first scenario has all daemons running so the bid can be returned quickly by all the remote machines and then evaluated by the source machine (this usually takes around 2 seconds). But suppose that one of the remote daemon processes is down because a machine had crashed. The daemon on the source machine must then timeout the bid process since the daemon on the crashed machine will never respond. We currently have the timeout set at 20 seconds so that if the network and other machines are heavily loaded we don't timeout the bidding process prematurely. We also noticed that there was no significant difference in the total time that the bidding process required when the number of candidate machines varied (even over a large range).

When users submit processes for remote execution by DAWGS, they see very little of what actually takes place. The time delay the user sees is the time for the front-end process to transmit all of the user's environment, terminal, and other information to fill in the queue structure (shown in Section 4.3) and was measured at 1.86 seconds. Once the daemon has this information, it can solicit bids and then migrate the process. After a bid had been selected, it took, on average, 4.89 seconds to move the process and associated information to the remote machine and is referred to as M_t. On average, the combined times from the time the user first invoked the front-end process to the time the process had been migrated to the remote machine and was ready for execution was 12.0 seconds. When the process was complete, it took, on average, 2.09 seconds to move all necessary data from the remote machine back to the source machine so it could be returned to the user and is referred to as R_t.

Another important function of the daemon process is to checkpoint processes running remotely once every α minutes. We measured the amount of time needed to checkpoint a file to be 1.36 seconds (this having created a checkpoint image of 18508 bytes). Most of the overhead in this number seems to be related to NFS work since all of our filesystems on the 6152 network were NFS-mounted (except for very small root and usr filesystems). We measured that our checkpoint file was being written to disk at 15 KBytes/second, so even for large programs the checkpoint process doesn't consume a large percentage of time and introduces a high degree of reliability.

When attempting to restart the checkpointed image, the first step of the daemon is to build a new a.out that the kernel can page from. We observed that the average time to build the new a.out file from the original executable and the checkpoint image was 0.76 seconds (which resulted in an a.out size of 11272 bytes, in contrast to the original a.out size of 14336 bytes). Similar to the checkpoint file, the a.out file was built at an average of 15K/sec. In the new a.out executable, the symbol table, string table, and relocation information are stripped from the new executable (similar to the Unix command *strip*).

6.3 Expected Run-time of Remote Processes

When attempting to run processes remotely, there is a point where the expected run time of the process running locally becomes greater than the run time of the process if it were run remotely. This is defined as the *break-even point*, or the point at which it actually is better to run the process remotely than to run the process on the local machine. In attempting to describe the run-time of the process, we define x to be the actual run-time of the process (in seconds) supposing that it were placed on a machine and given exclusive use of the CPU until it had finished executing.

For the expected local run-time of the process, we define the equation

$$E_l(r)=x+y \tag{1}$$

where y is the local overhead which will impede the progress of the process to completion. To describe the expected run-time of the remote-running process, we define the equation

$$E_r(r)=x+B_t+M_t+R_t+\frac{x}{\alpha}C_t+E_{rd}(d) \tag{2}$$

which is essentially the time for the process to run, the time required for bidding, migrating the process, and returning the results, the time needed to checkpoint the process (if required), and the time needed to find a new remote machine if the original remote machine suffers a idle-to-busy transition and returns the process before the process has run to completion. The expected value of the time that's incurred for a delay due to a user reclaiming a workstation is:

$$\begin{aligned} E_{rd}(d)=&E(reclaim\_delay \mid reclaim)Prob[reclaim]+ \\ &E(reclaim\_delay \mid noreclaim)Prob[noreclaim]+ \\ &E_{rd}(d)Prob[noreclaim] \end{aligned} \tag{3}$$

This equation specifies the time that we expect to be delayed (in the dawghouse) given that a reclaim has occurred plus the time it will take to move a process from the dawghouse back to the source machine, rebid, and migrate it to a new remote machine given that no reclaim has occurred plus the time needed to migrate again if the new remote machine has an idle-to-busy transition and another forced migration occurs. We assume that there is always at least one idle workstation (something which we noticed happened more than 95% of the time, so its a reasonable assumption). This equation can be simplified to

$$E_{rd}(d)=\frac{Prob[reclaim](\beta+T_m Prob[T>\beta])}{1-Prob[reclaim]} \tag{4}$$

where T_m is the time to migrate the process back to the source machine, re-bid, and then migrate the process to a new remote machine. Placing the known values of β=20 minutes, T_m=15.03 seconds, and the Prob[T>β]=.514 into the equation, we then get

$$E_{rd}(d)=\frac{Prob[reclaim]624.525}{1-Prob[reclaim]} \tag{5}$$

Now we try to find the point where remote execution will be faster than local execution. We have plotted these times in Figure 5. The solid lines represent local execution times of 1 min., 5 min., 20 min., and 1 hr. 20 min., and the dotted lines the corresponding remote execution times. This graph shows that remote execution can be

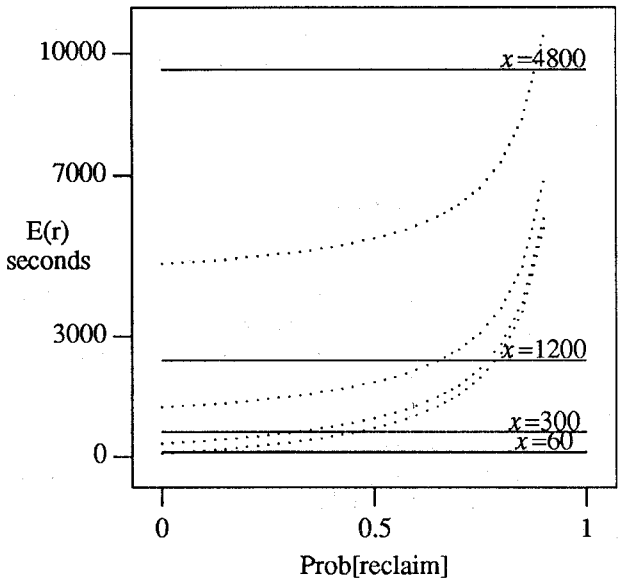

Figure 5: Remote Execution Times

substantially better than local execution, particularly for long-running jobs. But we also observe that the more a process is migrated, its execution time will tend to increase exponentially. Now we show in Figure 6 the effect of freezing the probability of the reclamation of a workstation by a user and compare the overall time needed for execution of the process.

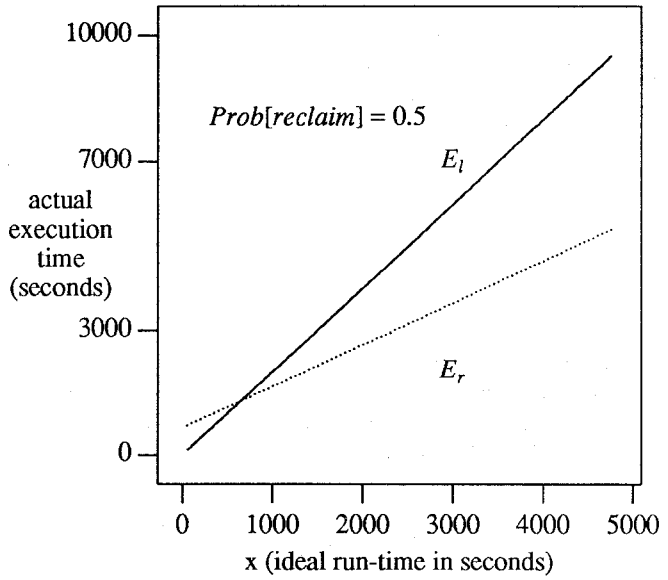

Figure 6: Fixed Probability Runtimes

As we can see in the graph, remote execution pays off better than we had expected. Even assuming a probability of forced migration of 50%, we still achieve a *guaranteed* remote speed-up for any jobs that runs longer than 10 min. 39 sec. From Figure 5 we can see that if we assumed a lower probability of forced migration, we could see even better results for the break-even point.

We assume that y is equal to x in equation 1. Is this a valid assumption? Assume that we're running a compute-bound process locally. The CPU scheduler is going to interrupt the process after its time quantum has expired and give the CPU to another process that's waiting to run. Since the user is using the local machine for other things (like reading mail, editing files, etc.), making the assumption that the process will take twice as long to complete as running it on a remote idle CPU is valid (of course, it could take more than twice as long to complete in a worst-case scenario).

7.0 Conclusions

DAWGS seems to be a useful tool to allow users to utilize the idle time of workstations on a local area network for their own computations. Users who want to run additional processes are provided with an easy to use tool which forces them to use no special system calls or know anything about the execution of their process. Non-trivial processes which run under DAWGS will see speed-ups in their run-time as compared to local execution. We currently use DAWGS to run coarse-grained parallel jobs with no interaction.

In the future, socket-handling capability will be added to DAWGS. Another interesting area of research is to add the capability of true *load balancing* to DAWGS. The current version of DAWGS was built to utilize idle workstations, but not to do load balancing in the true sense of the term. Since DAWGS has shown that processes can be transparently moved at will, future attempts at providing load balancing, both for submitted processes to be run remotely and for other user processes, may be promising.

References

[1] David A. Nichols, "Using Idle Workstations in a Shared Computing Environment," *Operating Systems Review*, November 1987, pp. 5-12.

[2] Michael J. Litzkow, Miron Livny, and Matt W. Mutka, "Condor - A Hunter of Idle Workstations," *Proceedings of the 1988 Conference on Distributed Computing Systems*, pp. 104-111.

[3] Robert Hagmann, "Process Server: Sharing Processing Power in a Workstation Environment," *Proceedings of the 1986 Conference on Distributed Computing Systems*, pp. 260-267.

[4] Lionel M. Ni, Chong-Wei Xu, and Thomas B. Gendreau, "A Distributed Drafting Algorithm for Load Balancing," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 10, October 1985,

pp. 1153-1161.

[5] John A. Stankovic and Inderjit S. Sidhu, "An Adaptive Bidding Algorithm for Processes, Clusters, and Distributed Groups," *Proceedings of the 1984 Conference on Distributed Computing Systems*, pp. 49-59.

[6] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel, "The LOCUS Distributed Operating System," *Proceedings of the Ninth Symposium on Operating Systems Principles*, October 1983, pp. 49-70.

[7] M.L. Powell and B.P. Miller, "Process Migration in DEMOS/MP," *Proceedings of the Ninth Symposium on Operating System Principles*, October 1983, pp. 110-119.

[8] David J. DeWitt, Raphael Finkel, Marvin Soloman, "The CRYSTAL Multicomputer System," *IEEE Transactions on Software Engineering*, Vol. SE-18, Number 8, August 1987, pp. 953-966.

[9] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proceedings of the Tenth Symposium on Operating System Principles*, December 1985, pp. 2-12.

[10] Michael J. Litzkow, "Remote Unix: Turning Idle Workstations into Cycle Servers," *Proceedings of the 1987 Summer USENIX Conference*, Phoenix, AZ, pp. 381-384.

[11] Katherine M. Baumgartner, Ralph M. Kling, Benjamin W. Wah, "Implementation of GAMMON: An Efficient Load-Balancing Strategy for a Local Computer System," *Proceedings of the 1989 Conference on Parallel Processing*, 1989, pp. II-77 - II- 80.

[12] Marvin M. Theimer and Keith A. Lantz, "Finding Idle Machines in a Workstation-based Distributed System," *Proceedings of the 1988 Conference on Distributed Computing Systems*, 1988, pp. 112-122.

[13] Phillip Krueger and Miron Livny, "A Comparison of Preemptive and Non-Preemptive Load Distributing," *Proceedings of the 1988 Conference on Distributed Computing Systems*, 1988, pp. 123-130.

[14] Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System,*, Addison-Wesley, 1989, pp. 72-79.

[15] Andrew S. Tanenbaum and Robbert Van Renesse, "Distributed Computing Surveys," *ACM Computing Surveys*, Vol. 17, No. 4, December 1985, pp. 419-470.

[16] Raymond M. Bryant and Raphael A. Finkel, "A Stable Distributed Scheduling Algorithm," *Proceedings of the 1981 Conference on Distributed Computing Systems*, pp. 314-323.

[17] Ami Litman, "The DUNIX Distributed Operating System," *Operating Systems Review*, XXXXXXX 198X, pp. 42-51.

[18] Jo-Mei Chung and N. F. Maxemchuk, "Reliable Broadcast Protocols," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, Vol. 2, No. 3, August 1984, pp. 251-273.

[19] Michael L. Powell and David Prescotte, "Publishing: A Reliable Broadcast Communication Mechanism," *Proceedings of the Ninth Symposium Operating Systems Principles*, October 1983, pp. 100-109.

[20] Edward R. Zayas, "Attacking the Process Migration Bottleneck", *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, Vol. 21, No. 5, pp. 13-24.

[21] Timothy C. K. Chou and Jacob A. Abraham, "Load Balancing in Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, July 1982, pp. 401-412.

[22] Craig E. Wells, "A Service Execution Mechanism for a Distributed Environment," *Proceedings of the 1989 Conference on Distributed Computing Systems*, pp. 326-333.

[23] Leonard Kleinrock and Willard Korfhage, "Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems," *Proceedings of the 1989 Conference on Distributed Computing Systems*, pp. 482-489.

[24] Kemel Efe and Bojan Groselj, "Minimizing Control Overheads in Adaptive Load Sharing," *IProceedings of the 1989 Conference on Distributed Computing Systems*, pp. 307-315.

[25] Ravi Mirchandaney, Don Towsley, and John Stankovic, "Adaptive Load Sharing in Heterogeneous Systems," *Proceedings of the 1989 Conference on Distributed Computing Systems*, pp. 298-306.

[26] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum, "The Performance of the Amoeba Distributed Operating System," *Software Practice and Experience*, vol. 19, March 1989, pp. 223-234.