Overhauling Rdist for the '90s

Michael A. Cooper − University of Southern California

ABSTRACT

The *rdist* program was first released as part of 4.3BSD UNIX. It was one of the first programs to address the area of automated software distribution in a timely, consistent manner in a highly distributed environment. Since its first release it has gained very wide use on almost every major UNIX platform. Despite all of its initial advances, *rdist* has some deficiencies which have not been addressed in any public release of the software until now.

This paper describes the past, present, and future of *rdist*. The program's history, operation, comparisons to other similar packages, and a number of possible future improvements in performance and functionality are described. A detailed description of *rdist* version 6, a major new version of the software that will be included in 4.4BSD, is presented. Version 6 *rdist* addresses many of the deficiencies of the original rdist, including significantly improved performance when updating large numbers of hosts, better error handling and avoidance, improved security, and cleanup of the actual source code.

Introduction

This paper describes major work done to a new version of *rdist* (version 6.0) that will appear in 4.4BSD. A look at the performance improvements in version 6, as well as possible future improvements, is discussed. The basic history and operation of *rdist* is described along with comparisons to other remote distribution packages.

The USC Environment

In order to provide better insight into the motivations behind the work described in this paper, it is helpful to describe the computing environment at the University of Southern California (USC) where most of the work was performed.

USC is a private University composed of two major campuses − University Park Campus (UPC) and Health Sciences Campus (HSC) − which are located some six miles apart.

The computing environment consists of about 3,000 hosts. This breaks down to about 1,000 UNIX machines (about 800 are Suns with the balance including NeXT, IBM RS6000, HP, and Silicon Graphics), 1,000 MS-DOS machines, 850 MacIntosh machines, and about 200 ''other'' types of machines. There are approximately 3,500 total ''nodes'' on USCnet, the campus LAN. USCnet physically consists of 12 cisco routers, 3 NSC routers, 1 Optical Data Systems (ODS) FDDI concentrator and myriads of various Ethernet and Fiber bridges and repeaters. Most of the routers are connected to an FDDI backbone. Twelve Sun SPARCsystems and one IBM RS6000 are connected to the ODS FDDI concentrator.

University Computing Services (UCS) is a service organization at USC which is charged with providing computing resources and support to the USC community. UCS centrally manages both centralized and distributed computing facilities. Approximately 800 UNIX hosts at USC are supported by UCS. Additionally, USCnet is also ''owned'', operated, and maintained by UCS. While UCS supports machines that are physically located in a centralized machine room, most computing resources are widely dispersed across the UPC and HSC campuses.

The current UCS support model for UNIX hosts is that all distributed and centralized resources are managed directly by a small group of people operating from a centralized physical location. Almost all major system administration work is done by UCS instead of by local system administrators. Each departmental machine under support has a designated Technical Contact. This person is usually a graduate student or professor who is responsible for performing ''day-to-day'' local system administration, such as resetting printer queues, setting up user accounts, and maintaining user disk quotas. UCS is usually the only party that has ''root'' (super-user) on machines connected to USCnet. The Technical Contact is provided with a special account which has access to a **set-uid** to ''root'' program which can run certain other programs as any valid user or group.

When problems or requests arise that are outside of the ''day-to-day'' privileges allowed the Technical Contact, they contact UCS via an electronic mail address called *action* or by calling the UCS Customer Service number.

History

History of rdist

Rdist was originally written in 1983-1984 by Ralph Campbell, then a graduate student at the University of California Berkeley. The only major release of the software was the version included in the 4.3BSD UNIX release in 1986. Since then, several major and minor security problems were fixed by the CSRG group at Berkeley. Various sites and individuals in the research community have also modified *rdist* to increase functionality and performance. Those modifications, plus a number of bugs and suggestions submitted to CSRG, have been evaluated and addressed in *rdist* 6.0.

History of rdist at USC

The first experiences with UNIX at USC where with a couple of DEC VAX-11/750's running 4.2BSD UNIX back in 1984-1985. It was apparent from the start that some type of software was needed to maintain distributed computing resources. However, nothing was available at that time that really addressed the issues involved with maintaining thousands and thousands of files on a large number of machines (two dozen UNIX hosts was considered a ''large number'' back then). When we purchased our first workstations (Sun-3/50's) in early 1986, there was still nothing to maintain all the distributed files. We were still muddling along with simple shell scripts for our thirty UNIX hosts. By the time *rdist* was released with 4.3BSD, and shortly afterwards with SunOS, we were really starting to struggle to maintain our 100 UNIX hosts.

The release and subsequent Great Discovery of *rdist* in 1986-1987 came as the number of UNIX workstations started doubling every year. Over the year or so following the Great Discovery, the number of supported machines continued to dramatically increase. *Rdist* all too quickly started showing signs of significant deficiencies in performance. The goals of the part time work that started in 1988, were to improve efficiency, reliability, and performance, in order to support the ever growing number and variety of UNIX machines. Looking back at all the work done, it would have been far better in the long run to have completely re-written *rdist* from scratch.[1]

The first major change made was to add support for updating multiple hosts in parallel. Despite a large number of other changes since then, this one item still remains as the single largest performance increase realized to date.

Release History

The first version of *rdist* to be released was the version included in 4.3BSD. This version spoke version 3 of the *rdist* protocol.

The first version of *rdist* to be released externally from USC was version 5.0 (protocol version 5) in 1991. This version never got farther then *beta* testing at a small number of external sites. During the *beta* test, a copy of all the submissions regarding *rdist* bugs, suggestions, and improvements sent to the CSRG group at UC Berkeley, was received. This resulted in a significant number of changes that obsoleted version 5.0 and spawned version 6.0 (protocol version 6), which is the first version of *rdist* from USC to make it into general distribution.

Description of Rdist

What It Does

Rdist is a program that maintains identical copies of files over multiple hosts. It preserves the owner, group, mode, and modification time (*mtime*) of files, if possible, and can update programs that are executing. *Rdist* reads commands from a *distfile* to direct the updating of files and/or directories similar to how *make* reads recipes from a *Makefile*. The syntax of *distfile* is described in the *rdist*(1) man page in Appendix A.

What It's Good For

Rdist is a general purpose program which can be utilized for a number of purposes. It provides an excellent mechanism for maintaining consistent versions of files that contain text, binary, or other type of data. This is invaluable for maintaining the same version of operating system files on a multitude of hosts.

A number of sites use *rdist* in place of Sun's Network Information Service (NIS, aka YP) to maintain consistent, distributed copies of */etc/hosts*, */etc/passwd*, */etc/group*, */etc/services*, and others ([Satd91a]).

At USC, *rdist* is normally used to maintain consistent versions of operating system files as well as third party software packages such as X11, GNU Emacs, and many others. It is often used to perform minor operating system upgrades such as upgrading hosts from SunOS 4.1.1 to SunOS 4.1.2.

How It Works

Rdist parses a file called *distfile* which contains a description of what needs to be done. Once this file is parsed, *rdist* will open connections to multiple clients in parallel using the *rcmd*(3) interface[2]. If a connection attempt fails, or a running session fails, no further connection attempts are made during that instance of *rdist* in order to avoid slowing down updates to other hosts.

The *rcmd*() routine makes a connection to the *rshd*(8c) program on the remote host. The *rshd* program in turns checks to see if the user is authorized via the ''.rhosts'' mechanism. If the authorization succeeds, then the user's login shell is started, using the command:

`rdistd -S`[3]

The user's login shell searches for the program

[1] Hind sight is always 20/20.

[2] The *rcmd*(3) interface is the same one used by the *rsh*(1c) command.

[3] The **−S** option is required to avoid accidental execution by actual users since *rdistd* usually resides in the normal user's search path.

rdistd in the user's search path (**$PATH**) and executes it if found. *Rdistd* now sends a message back to *rdist*, through the *rsh* connection, requesting *rdist* to send its protocol version number. *Rdistd* reads the version number and decides whether it supports that protocol version. If it cannot support it, it will return an appropriate error message to *rdist* and exit. Otherwise *rdistd* starts accepting commands from *rdist*.

After negotiating the protocol version with *rdistd, rdist* now sends global configuration information to *rdistd*. This information includes the name of the host *rdist* is running on[4] as well as any global parameters such as minimum amount of free space and files that must exist before a file will be installed. Once this configuration information is successfully sent, *rdist* starts checking files as specified in the *distfile*. The actual protocol involved in this, is beyond the scope of this paper.

Methodology of Operation

The methodology behind the operation of *rdist* is one of a dictator forcing his/her view upon his/her subjects regardless of the subject's will. This works well in environments where there is centralized management which has almost absolute control over the distributed environment. A centralized machine (or machines) can utilize *rdist* to maintain a large number of distributed machines.

Another major benefit to this model is in security. Maintaining a tight security leash on a few centralized machines which *rdist* to some number of other machines can be an excellent mechanism for automatic detection and correction of security intrusions. The corruption of system programs is a typical method used by many intruders to gain further access and privileges. This type of attack is easily detected and corrected with the proper application of *rdist*. Obviously, the master *rdist* host(s) must be carefully guarded against being compromised, or else all your client machines may become compromised.

Comparisons

Comparison to Track

The *track* system as described in [Nach86a] is another automatic software distribution package similar in capabilities to *rdist*. It differs significantly in it's basic architecture and the methodology behind it.

[4] *Rdistd* does not know the name of the host *rdist* is running on because *rshd* does not provide any mechanism for obtaining this information. Thus, the host name is for informational use only and should never be used for any kind of authentication.

Architecture Differences

The main architectural difference between *rdist* and *track* is in the *track* model of *pulling* files from a centralized *Librarian* host down to *Subscriber* hosts. The *Subscriber* host has a *subscription* file which contains lists of software *packages* and the *Library* host that they reside on. The *Subscriber* host is responsible for initiating contact with the appropriate *Library* host(s) in order to update and/or verify *packages.*

This model works well in environments where distributed hosts are administered by mostly autonomous local system administrators. In environments where centralized control and security are of primary importance, the *rdist* model of *pushing* files out to client machines is more in line with those priorities.

With *rdist*, the *master* distribution host is the only point where configuration information must be maintained. The *track* model is that each *Subscriber* must maintain the list of packages it wants to receive. The *Librarian* maintains the details of each package, which includes the package contents and the *Subscribers* authorized to receive the package. A misconfigured *Subscriber* host may go undetected for a long period of time. Automating the distribution of the subscription information from a central *Library* host may solve some of the problems. However, this would remove the local autonomy aspect of *track* and still not provide for the automatic detection and correction of security intrusions that is inherent in the architecture of *rdist*.

Transport

Track also differs from *rdist* in the transport layer used to communicate between client and server. *Rdist* uses *rsh* (via the *rcmd*() library routine) as its transport layer. *Rsh* in turn is implemented on top of TCP/IP. *Rdist* does **NO** authentication itself, instead relying on *rsh* to perform the authentication. *Track*, on the other hand, is implemented directly on top of the TCP/IP layer. The client and server communication through a ''well-known'', privileged TCP port. The server is responsible for doing all authentication.

One of the benefits of using *rsh* as the transport layer, is in portability to other platforms and network protocols. Since most every 4.X BSD UNIX based platform supports *rsh* there is no transport layer in *rdist* to port. If *rsh* is ported to utilize another network protocol, *rdist* can also automatically make use of it.

On the down side, using *rsh* as a transport layer severely limits the authentication performed for an application. Currently *rsh* relies on the often insecure method of IP host address and privileged ports as a means of authentication. This does not provide for a secure means of *rdistd* obtaining the actual identity of the user or host running *rdist* on the remote host.

The implication here is that *rdist* cannot provide a *subscription* service as *track* does. The reason for this lies in *rsh* which hides the actual identity of the caller. This means *Rdistd* doesn't know the authenticated identity of the caller, which is a requirement of most *subscription* oriented services in order to restrict access of certain files and commands to certain callers. *Track* can authenticate the caller's identity since it listens on its own TCP port and therefore ''knows'' the identity of its caller in the same manner that *rsh* does.

The eventual solution to the caller identification problem in *rdist* is probably *Kerberos*. Currently *rdist* does not use *Kerberos* since there are still a few vulnerabilities in the *Kerberos* protocol and because its use is not yet wide spread. It should be a trivial matter to add *Kerberos* support to *rdist* at a later time.

Changing *rdist* to eliminate the *rsh* layer and use its own well-known TCP port was considered at one point. The only major benefit gained would be the ability to implement a subscription service. The amount of work and additional complexity required for this was deemed excessive in the face of the eventual use of *Kerberos*. Those sites requiring the *pull file* model can use *track*. Those sites requiring the *push file* model can use *rdist* as currently implemented.

Feature Differences

There are some basic feature differences between *rdist* and *track*. One of the most notable is *track*'s use of a *state database* on the *Librarian* host, which is implemented as a ''flat'' ASCII text file. The database contains state information for files in the *Library*. This information is basically the file type, the path name of the file, and the *currentness* of the file. The *currentness* is defined based on the type of file. The *currentness* of regular files is the file's modification time. For directories its based on the owner and group of the directory. For symbolic links it is the pathname the link points to.

The state database is periodically generated by running a command either via an automatic scheduler such as *cron*, or manually from a users shell. The *track* state database generator provides for only coarse updates by checking all the files listed in the *Library* subscription list. This model works sufficiently if updates of files that are in the *Library* are infrequent. In a volatile environment such as USC, which has frequent (daily) updates to many different ''sections'' of a *Library*, a coarse grain state database generator involves significant overhead which greatly increases the time to update files. Experiments with using the same type of state database generator with *rdist* showed an overall *increase* in the time it took to verify a large number of *volatile* files.

The *rdist* master host at USC which updates the most files is a Sun SPARCsystem 490 with 128MB main memory and 15GB of IPI-2 disk. Using a coarse grain generator similar to *track*'s, it took approximately 8.3 hours to generate a database for some 512,000 files. Once generated, an *rdist* session reading from this database instead of doing a *stat*() call for every file for every client, took 3.3 hours to do an *rdist verify* of all files on 103 hosts, with 4 clients being checked simultaneously. The same setup *without* using the state database took 4.5 hours. The total aggregate time with the state database is 11.6 hours compared to 4.5 hours without. Notice that the *without* number is not significantly higher than the verify time of the *with* number. This is most likely due to the kernel caching the inode information which is advantageous to the simultaneous updates of clients since they are usually checking the same files at roughly the same time. A possible solution to this problem is to implement a fine grain state database generator which is discussed later in the paper in the **Future Directions** section.

Other Differences

Other features that are present in *rdist* but missing in *track* include *track's* inability to detect when the owner, group, file mode, and/or contents of a regular file has changed, but the modification time (*mtime*) has not. Only the *mtime* is checked to determine if the file is current or not.

Track also lacks a regular expression facility for the *exception* field in the *subscription* file. This could be corrected by using one of the many regular expression libraries available, such as the Berkeley regex (*re_comp()/re_exec()*) routines which are used by *rdist*.

Comparison to Ninstall

Ninstall as described in [Rodr87a] is similar in architecture to *track*. It provides the same model for *pulling* files from other hosts and installing them on the machine it was run on. It differs from *rdist* along the same lines as *track*.

Comparison to Ru

Ru as described in [Sigm87a] is also similar in concept to *track*. It differs in architecture with *rdist* in a manner similar to *track*. The fact that it's implemented as a small set of shell scripts shows that an automated software distribution system need not be complex to provide minimal functionality.

Deficiencies of Original Rdist

The original *rdist* had a number of deficiencies that are described below. It should be noted that for all it's flaws, it made for a very useful and pioneering tool which helped pave the way for *rdist* version 6 and several others.

Serial Host Updates

The original *rdist* would only update one single host at a time. Updating a large number of hosts was extremely slow and inefficient because of this. One very slow host, or a connection that hangs, would also delay updates for all other hosts, possibly indefinitely.

In 1983, when *rdist* was originally written, CPU speed and I/O bandwidth was still relatively slow and kernel *inode* caching techniques were somewhat primitive. This made applications such as *rdist* extremely disk bound in performance. The benefits of parallel *rdist* updates were negligible when faced with these limits. Since the original version was written, significant increases in CPU speed and I/O bandwidth, as well as advances in kernel *inode* caching techniques, have made parallelizing once disk-bound applications, such as *rdist*, very appealing.

Hangs

The original *rdist* never attempted to avoid hanging. A remote host could get into a state where it never completed an *rdist* session, thus hanging all other host updates.

Security Problems

The original *rdist* had one program, *rdist,* which was **set-uid** to ''root''. This meant that both the client and server *rdist* processes had to run as ''root'' which led to a number of security holes.

Bad Coding Style

The coding style used in the original *rdist* left even some of the best C programmers mystified as to many of the its inner workings. One of the most notable problems was in the use of hardwired protocol values. Instead of using *#define* definitions for the *rdist* protocol commands, the actual values were coded in multiple places (usually at least twice − once in the client modules and once in the server modules) throughout the source code. This made following, let alone extending, the protocol logic a feat not for the meek.

The repetition of redundant code often served to cause confusion when changes were made to one part of the code, but not another. The nonuse of ''standard'' C library routines also led the code being to illegible, redundant, and non-portable.

The combination of the client and server programs into one program made the logic flow cluttered and unnecessarily complex in many areas. This often resulted in vulnerabilities in security.

The rampant use of *goto*s and use of large routines with many levels instead of smaller, simpler functions, also made logic flow unnecessarily difficult to follow.

Miscellaneous

The mode and ownership of files was never checked for *currentness* in the original *rdist*. Only when a file's *mtime* was found to be out of date and a file was updated, was the file's mode and ownership set.

The original *rdist* could not handle rdisting a directory on the local machine to a symlink on a remote machine. e.g. Suppose you have a directory */foo/bar* on machine A and machine B. If you replace */foo/bar* on machine B with a symlink and then *rdist* from machine A to machine B, *rdist* would fail (with an error) to update */foo/bar* on machine B.

New Features of Rdist

Many new features have been added to *rdist* 6.0 since the release of the original in 4.3BSD. Some of the highlights are detailed below.

Parallel Host Updates

Multiple target hosts are now updated in parallel. This can dramatically improve the update time for a large number of hosts. The effective number of concurrent updates is limited by the CPU and I/O bandwidth of the host running *rdist*. The value of 4 concurrent updates has proven an adequate default value for many different platforms. See the **Performance** section for a discussion of this.

Improved Error Handling and Avoidance

Timeouts

There are a number of ways that *rdist* can hang if not properly avoided. One such case is when a host goes down and does not reboot. If this occurs while *rdist* is doing a *read*() waiting for response from the unreachable host, the TCP/IP kernel layer does not pass up a failure error unless the remote host comes back up on the network. A similar situation can occur if the remote host is a NIS client and loses contact with an NIS server.

Rdist avoids hanging by setting timeout alarms via *alarm*() and *signal*(). Before calling *read*() to read a response from the remote host, an *alarm*() is set for a given period of time (10 minutes by default). If no response has been received at the end of the timeout period, the session is marked as *failed* and the child *rdist* process for that particular host exits with an error status. The parent *rdist* process then proceeds to the next host to update.

Avoid Retries

If a connection to a host fails at any point for any reason, no further attempts are made to contact the host during that instance of *rdist*. This avoids having a host that is unreachable, or not responding properly, from slowing down, or even preventing, the updates of other hosts.

Error Messages

Local and remote error messages are distinctly marked as such for better clarity as to the origin of the error. This makes determining the cause of the error much simpler.

Free Space Checking

The amount of free space and/or free files/*inodes* can optionally be checked to avoid filling up a filesystem. Before actually installing or updating a file, *Rdist* will calculate whether the update would exceed the minimum amount of free space and/or inodes as specified on the command line. If the minimum space would be exceeded by the update, no update is performed and an error message is displayed. This allows *rdist* to be less intrusive by preventing it from filling up a filesystem.

The cost for this avoidance is increased overhead resulting from additional system calls. The *statfs*() system call is called for most every update of a file. The calculation and call are performed after the *currentness* check but before the actual install/update is performed.

Split Client and Server

The client and server portions have been split into two distinct programs, *rdist* and *rdistd*, respectively. This lowers the risk of security vulnerabilities since the server *rdistd*, does not need to be **setuid** to ‘‘root’’. It also allows for greater ease in maintaining different protocol versions of *rdist*.

Major Code Cleanup

Most sections of the source code have been cleaned up, and in many cases, re-written. A majority of this cleanup was to make the code more understandable and to clarify the underlying *rdist* protocol itself. Many routines were converted to use standard system library routines. The code has been ported to a number of different UNIX platforms, including several System V.3 based systems, with relatively little changes.

Miscellaneous

All reported security holes have been fixed.

Modes of files and directories are now checked. If they are different, the entire file on the remote machine is updated. This is done instead of just changing the mode and/or ownership in case the file was compromised in some manner. This is usually quicker than doing a full binary comparison of the local and remote files.

Rdist now has the ability to rdist a directory to a symbolic link. The original rdist could not handle this.

Rdist will optionally check whether a file resides on a NFS and/or a read-only filesystem. If so, no update will be done unless explicitly specified for that host.

A general protocol command was added during the initial connection negotiation to support setting certain parameters. These parameters include the name of the host running *rdist*, the minimum amount of free space, and the minimum number of free files/*inodes* that must be available on a filesystem for a file to be updated. The host name is used for internal logging purposes by *rdistd* and is also used in setting the process arguments to show where the host is being rdist’ed from.

Support was added to do buffered reads. Instead of reading one character at a time from the remote host like the original *rdist* did, it now attempts to fill a full 8 kilobyte buffer whenever possible. This can lead to measurable reductions in system overhead due to the reduced number of kernel *read*() calls needed.

All configuration information is now stored in two places, **config.h** and **Makefile**. Porting to a new platform usually only requires slight modifications to these files.

Performance

Parallel Updates

The most significant performance improvement seen in version 6 of *rdist* is in parallel updates. This can be evaluated by running a series of tests varying the number of simultaneous updates. **Table 1** shows the results of two such tests. The **Number** column indicates the number of simultaneous clients being updated, **Time** indicates elapsed ‘‘wall-clock’’ time in minutes, and **%CPU** indicates the percent of the CPU used.

Table 1

Number	Test 1		Test 2	
	Time	%CPU	Time	%CPU
1	756	10	1600	18
2	309	24	1012	30
3	254	31	713	48
4	175	48	873	36
5	217	40	413	73
6	195	46	431	71
7	198	42	446	69
8	207	40	432	73
9	208	40	510	63
10	200	42	568	58

The test used in both cases was to run *rdist* in *verify* mode, varying the number of simultaneous clients being updated between 1 and 10. Each test used a different *rdist* master and a different set and number of files. All machines involved with the test, including the master *rdist* hosts, where running their normal work load. This accounts for some discrepancies in the results.

Results of Test 1

The *distfile* used in **Test 1** is used at USC to maintain a body of some 10GB of software. The test was run on a Sun SPARCsystem 490 with 128MB of RAM and 15GB of IPI-2 disk. A total of 29 hosts were listed in the *distfile* as client hosts.

The results for this test indicate that the optimum number of simultaneous updates is 4. The elapsed time drops off most significantly between 1 and 2 clients, in comparison to any of the other numbers. This is likely due to the kernel caching the *inode* information that occurs when *rdist* calls *stat*() which is where *rdist* incurs most of its overhead. Since the two *rdist* processes are *stat*()’ing the same set of files at roughly the same time, the kernel only has to read the *inode* information once from disk in most cases.

The amount of CPU used in **Test 1** levels off around 42% starting around 4 parallel updates. This may also be an indication that the kernel is able to use cached *inode* information most of the time. This ability is probably due to the large amount of RAM (128MB) in the master machine used for this test, which allows more information to be kept in the kernel *inode* cache.

Results of Test 2

Test 2 uses a *distfile* that is used at USC to maintain the SunOS 4.1.2 operating system files which include */sbin*, */usr/5bin*, */usr/5include*, */usr/5lib*, */usr/bin*, */usr/etc*, */usr/kvm*, */usr/share*, and */usr/ucb*. The *rdist* master host used was a Sun SPARCstation ELC with 8MB of RAM and 300MB of SCSI disk. A total of 245 hosts were listed in the *distfile* as client hosts.

The results of this test show that 5 simultaneous updates is the most optimal value for elapsed time. However, the percent of CPU required for 5 updates is 73% as opposed to just 36% for 4 parallel updates. This indicates the presence of a threshold in the system of some type. Most likely the threshold is the system running out of physical memory which results in increased paging rates and keeping less *inode* information cached in the kernel.

Future Work

There are a number of possible avenues for future work to follow which may lead to increased performance and functionality.

State Database

The implementation of a state database, and a corresponding fine grain state database generator, is a likely source for further performance improvements. The cost of updating the state database must be lower in overhead than simply checking the state of each file during the actual update session as is done in the current implementation. Such a generator must have the ability to check and update specific files, directories, or packages of software. In this way, a system administrator who updates a file or entire package of software, could manually run the generator specifying the newly updated files or packages to check. Additionally, the most frequently changing packages could be automatically checked hourly or daily with a full check of everything on a weekly basis.

Another area of interest for the state database problem is the underlying database itself. Flat files are very inefficient to update and search if they contain large numbers (large being greater than 100,000) of entries (files) and are very hard to update quickly in a fine grain manner.

The *dbm*(3) routines also do not scale well to large numbers of entries. The new Berkeley hash database package [Selt91a] may be one possible solution. There are certainly a number of commercial databases that could handle the data, but the financial cost is, of course, prohibitive.

Message Handling

Message handling is one area which requires some attention. The quantity of output from *rdist* can be quite voluminous as well as useless in content. The amount of output that can result in just one new software package, such as MIT’s X11R5, can be hundreds of thousands of lines. Human perusal of so much output is tedious, boring, and is usually ignored.

The destination of the messages could also be improved. Currently, all errors that occur are displayed in the normal output along with the normal messages. Major errors that *rdistd* encounters, are logged via the *syslog* facility.

The current *rdist* includes a filter written in *Perl* to put the output in a more human readable format. It does nothing to reduce the amount of data nor provide any filtering of the type of messages seen.

To address this problem, a comprehensive overhaul of the message handling system in *rdist* is needed. All messages need to be classified by type and assigned a severity level which should look something like those of *syslog*. The message handling system will also need to be changed to support a variety of logging destinations such as files, *syslog*, electronic mail, and perhaps a general purpose facility to allow piping to an arbitrary user specified command. A better filter program on top of all this could also increase the readability and amount of data shown.

File Checking

Another area for possible performance improvement is in checking actual file contents. The current *rdist* offers the option of doing a byte-by-byte comparison of files. This is implemented by having *rdist* send a full copy of a file over to *rdistd* and then doing a byte-by-byte comparison. In the case of large files and relatively slow or congested network links, this can be prohibitive. Implementing a checksum test would address this issue. A checksum would be performed for the files on the client and server by both *rdist* and *rdistd* and the results compared. The file would then only need to be transferred if it's out of date and *rdist* is in update mode.

Polling the Rdist Server

An interesting feature that was added to the original 4.3BSD *rdist* by the Ballistic Research Laboratories (BRL) was the ability of a client *rdist* host to *poll* an *rdist* master host. This allows the client to be rdisted to on-demand as opposed to when the master wants to.

The way it works is the client runs *rdist* with the *poll* option specifying the name of the master *rdist* host to contact. The client *rdist* then uses the same *rcmd*() interface as the master does to open a *rsh* connection to the master. The command run via *rcmd*() tells the *rdist* server to go into *polling* mode and also specifies the name of the client doing the poll. The master host then *reverses* the connection, parses the *distfile* in the current directory (which should be the home directory of the user the server is running as), and proceeds with a normal *rdist* session for just the polling host.

The polling feature is not present in version 6.0 of *rdist*. The changes from BRL are not easily incorporated into version 6.0 due to the separation of the client and server processes into separate programs. It's also not clear if this is really that desirable. In order for it to work, the server program, *rdistd*, probably needs to be **set-uid** to ''root'', which removes one of the more desirable features of version 6 *rdist*.

Conclusions

The system described above has gone through a number of internal releases since work began 4 years ago. It currently is used to support some 800 UNIX machines at USC. While many new features have been added since the original 4.3BSD version, no further significant performance gains have been realized since the initial support for parallel updates was added. Adding support for a state database and fine grain database generator hold hope for another leap in performance which we hope will be able to sustain the current growth rate of UNIX machines here at USC and elsewhere.

Availability

Rdist is available via anonymous ftp from host **usc.edu** as **/pub/rdist/rdist.tar.Z**.

References

Satd91a. Bjorn Satdeva and Paul M. Moriarty, ''Fdist: A Domain Based File Distribution System for Heterogeneous Environment,'' in *LISA V Conference Proceedings*, pp. 109-126, USENIX, San Diego, CA, September 30 - October 3, 1991.

Nach86a. Daniel Nachbar, ''When Network File Systems Aren't Enough: Automatic Software Distribution Revisited,'' in *USENIX Conference Proceedings*, pp. 159-171, USENIX, Atlanta, GA, Summer 1986.

Rodr87a. Mike Rodriquez, ''Software Distribution in a Network Environment,'' in *Large Installation System Administrators Workshop Proceedings*, p. 20, USENIX, Philadelphia, PA, April 9-10, 1987.

Sigm87a. Tim Sigmon, ''Automatic Software Distribution,'' in *Large Installation System Administrators Workshop Proceedings*, p. 21, USENIX, Philadelphia, PA, April 9-10, 1987.

Selt91a. Margo Seltzer and Ozan Yigit, ''A New Hashing Package for UNIX,'' in *USENIX Conference Proceedings*, pp. 173-184, USENIX, Dallas, TX, January 21-25, 1991.

Author Information

Michael Cooper has been working on UNIX systems for 10 years. He has worked in the Research, Development, and Systems Group of University Computing Services at the University of Southern California since 1985. Reach him via U.S. Mail at: University Computing Services University of Southern California Los Angeles, California, 90089-0251. Reach him electronically at mcooper@usc.edu .