filetsf: A File Transfer System Based on lpr/lpd

John Sellens – University of Waterloo

September 22, 1995

Abstract

In a distributed computing environment, it is often necessary to transfer files between machines for administrative or record keeping purposes. Most methods of file transfer either require full-scale trust by the recipient (e.g. *rdist*, *rcp*), the use of a "secret" password (e.g. FTP), or some level of pre-arrangement (e.g. *rdist*, *track*). This paper describes **filetsf**, a generic file transfer system built on top of the standard **lpr/lpd** system. **Filetsf**'s advantages include minimal configuration, the need for minimal trust between sender and recipient, ordering and spooling of file transfer requests, and the elimination of the need for shared passwords.

This paper was first published and presented at the Ninth USENIX Systems Administration Conference, LISA '95, in Monterey, California, September 17-22, 1995.

Introduction

The University of Waterloo is in the process of converting its administrative systems from legacy implementations on IBM's VS/1 operating system under VM/CMS and the PICK database system on UNIX, to implementations based on the Oracle RDBMS on UNIX database and application servers. As part of the ongoing conversion, we have a need for regular (and irregular) file transfers between machines, primarily from the VS/1 and PICK systems to the Oracle systems, but also between UNIX systems.

To date, we have used a number of approaches, each of which seems to have had drawbacks, primarily security and reliability related. We needed a system to allow file transfers between our systems that allowed us to limit each programmer's sphere of control i.e. we wanted to avoid giving fullscale access to all systems and servers, just to enable file transfers. We also needed a file transfer system that was reasonably portable, and worked on UNIX and non-UNIX operating systems.

The end result was the **filetsf** family of programs, that provides the ability to send and receive files between cooperating machines, is implemented on top of the Berkeley **lpr/lpd**[3] system, and which provides a suitable level of flexibility while retaining access controls and an acceptable level of security.

Previous Approaches

In the past, we have primarily used FTP[11] for file transfer – this meant that most programmers and operators knew the passwords to any accounts that needed to receive files. This was not a good thing, since shared passwords are no longer secret, and because it allowed general use access to accounts that shouldn't have been used for anything except file transfer. And, in the absence of digital signatures, FTP does not allow the recipient to identify the sender once a file has been delivered, making it potentially easy for legitimate data to get replaced, intentionally or not.

We implemented a simple FTP-based system using a single recipient userid (named `unifer`, for UNIX transfer), where all files sent to a machine would end up in a single directory, in a particular group, with group write permission on everything. This meant, of course, that any file transferred to a machine was fair game for anyone else to access. Most file transfers were reasonably well-controlled, but left-over, unclaimed, files tended to accumulate, and it's hard to make a claim that this system provided a reasonable level of security and auditability.

We also used *rdist*(1)[1] in a few instances, which either required setup and monitoring by the (limited number of) super-users, or which required `.rhosts` file access to the receiving account, with many of the same problems as with FTP. *rdist* also isn't available on our VM/CMS system, which meant that it could not be used for the bulk of

[1] An enhanced version of *rdist* is described in [5].

our file transfers. *rcp*(1) suffers from many of the same problems as *rdist*, while providing a less-reliable copy to the recipient machine (*rdist* uses temporary files to ensure that the destination file is always complete, *rcp* just clobbers any existing file, and can leave files incomplete if interrupted or the destination disk fills up).

We haven't investigated *track*[9], because we have no experience with it on campus, and because, like *rdist*, it requires a level of pre-arrangement that we wanted to avoid.[2] *track*, like *rdist*, is also geared towards the replication of a set of files on multiple machines – for file transfer, we need to be able to send a file and then remove the original.

None of these approaches allows for the queueing of file transfer requests if the recipient or network is down, although it is possible to periodically repeat the *rdist* or *track* commands. And they don't provide for an ordering of the files transferred, unless you name the files with indicative names, or use file modification times as an indicator. In our previous systems, if a series of files needed to be processed (e.g. batches of transactions), the files were typically sent one at a time, processed on the recipient machine, and an acknowledgement was returned to the sender (usually via a flag file) to indicate that it was safe to send the next file.

Desired Features

We identified the following features as desirable in a file transfer system. Many of these are convenience features, intended to help enable a more automated environment, with less manual intervention (e.g. through avoiding entering passwords, or having to manually re-initiate transfers in case of failure).

- Available on all or most of our systems.
- Requires minimal configuration – we wanted and needed to be able to do ad-hoc file transfer.
- Allows file transfers between random pairs of users – root-to-root file transfers would not be enough, since we need to allow our programmers to use the file transfer system, none of whom are super-users.
- Allows the recipient to identify the sender and name of the file – a recipient could be receiving multiple copies of identically named files from more than one sender. For example, similar systems on different machines might generate transaction files to feed to a central server.
- Allows an ordering of files – e.g. files of transactions should be receivable in the order in which they were sent.
- Provides for queueing of file transfer requests, in case of machine or network unavailability. We didn't want to have to keep retrying file transfers manually if the destination machine was down or unavailable.
- Provides for a non-clashing name space – we wanted to be able to send multiple files with the same name, distinguished by sender, recipient, and/or time.
- Does not allow any additional access to the sending or receiving machine. We can't allow users, applications, operators or programmers to have access to anything more than they need to have access to.
- Provides a reasonable level of security in our environment. For most of our transfers, elaborate security is not required; we have a reasonable level of confidence in the security of our systems and networks. Encryption and digital signatures were not required in the basic file transfer system.
- Is based on file transfer, not file replication. We need to be able to generate a file, send it, and then destroy the original.
- Can operate unattended. Too many of our current systems require manual intervention from an operator, and there is increasing pressure (fiscal and otherwise) to avoid the need for manual processes in our systems.

VM/CMS has the *sendfile* command, which allows file transfer between users on VM/CMS systems – the file shows up in the recipient's "reader", tagged with the sender, filename, and file attributes. It meets many of our requirements, but we could not use it – we had no way to send the files to our UNIX machines from our VM/CMS machine, and we didn't have a VM/CMS compatible *sendfile* command (or equivalent) on our UNIX machines.

Approaches Considered

We were not aware of any publically available file transfer systems, and were trying to avoid investing in (and investigating) commercial solutions that might exist. We felt

[2] We may be investigating *track* for use in software package distribution, under our **xhier**[12] software maintenance system.

that our best approach would be to implement our own file transfer system.

Past LISA proceedings have contained quite a few papers on software distribution (xhier[12], depot[7], lude[6], lfu[1], CMU depot[4, 13], etc.), typically based on the idea of software "packages" that are replicated from master machines to slave machines, typically using *rdist*, NFS mounts, or FTP. These systems don't meet our needs, since they are based on the idea of replication of pre-defined sets, not the transfer of individual files. The tools used to transfer files (*rdist* and FTP) are not suited for our needs (as described above), and we were unwilling to investigate the use of NFS mounts, since that opens up a whole different can of worms, and since NFS is not available on our VM/CMS system.

Early in 1993, we started planning the construction of a complete file transfer system, with authentication, queueing, and so on, but we didn't make much headway, for a variety of reasons. That project never received much attention, and was eventually abandoned, and was set aside until a better opportunity presented itself.

It often seems to be the case that a problem left to its own devices eventually presents its own solution, and we finally realized, after much time had passed, that it would be relatively simple to implement a file transfer system on top of **lpr/lpd**. The **lpr/lpd** system has much of what we required, and it turned out to be relatively easy to layer the additional functions on top of **lpr/lpd** to obtain a file transfer system that met our needs.

Why not implement a file transfer system on top of SMTP[10] mail and MIME[2] file encapsulation? Mail doesn't meet some of our needs: it's too easy to forge, and it doesn't provide a reliable ordering of the messages. Digital signatures (such as those provided by PGP[14]) could be used, but they add another level of complexity, and may be harder to port to our non-UNIX systems. In addition, the path a mail message takes can't always be controlled (in the presence of MX records in the domain name server), and some mailers implement limits on the size of a message.

Why Use lpr/lpd?

It turned out that **lpr/lpd** and the LPD protocol[8] were a pretty good basis for a file transfer system. The protocol is well-defined, and widely implemented, and **lpr/lpd** already has queuing, file ordering, sender identification, machine-based access control, and so on built-in. The destination machine can be indicated by the "printer" name used, an **lpr/lpd** "output filter" can be used to process the files at the destination machine, and the job and class options to *lpr* can be used to identify the name of the file, and the intended recipient.

Using **lpr/lpd** was appealing because it seemed that it would save us a lot of effort – most of the hard stuff was already done – and because we could claim that we were behaving in a manner that was consistent with the "UNIX philosophy": small, simple, reusable tools put together to do more interesting things.

It was fairly easy to implement **filetsf** on top of **lpr/lpd**. The *sendfile* command is just a cover for *lpr*, *tsfif* is an **lpr/lpd** output filter that deposits the files into the **filetsf** spool directory, and we just have to set up a "sending" print queue for each machine we want to send to, and a a single "receiving" queue on each "destination" machine. In our case, the number of machines that we want to be able to send files to is manageably small, and our `printcap` file maintenance tools help a lot. An *lpr* client program[3] could be used to reduce the number of "sending" queues required, but this would mean that there would be no sender-side queueing, which is one of the features that we were interested in. One non-trivial program is required – *acceptfile* is a setgid program used to query and retrieve files from the **filetsf** spool directory.

One final indication that **lpr/lpd** and the LPD protocol were a good choice was the relative ease with which we could implement **filetsf** on our AIX systems (on top of IBM's queueing system, which is willing to use the LPD protocol to talk to other machines) and implement the sending side on our VM/CMS machine with a REXX script on top of the *netprint* command. We haven't investigated implementing the recipient side on VM/CMS yet, but we're optimistic that it shouldn't be too difficult, perhaps requiring just another REXX script running in a virtual machine invoking the VM/CMS *sendfile* command.

Implementation Details

Filetsf works by accepting files to send to a recipient, submitting them to **lpr/lpd**, delivering them into a spool directory, and waiting for the intended recipient to come and pick them up. A separate **filetsf** spool directory is used for easier access and control, and to provide a level of abstraction above that of particular **lpr/lpd** implementations. It also makes permissions easier to control, and

[3] One *lpr* client program is available from Keith Moore of the University of Tennessee as *ftp://cs.utk.edu/pub/moore/port-lpr.tar*

keeps our monitoring software from complaining that the print queues are stalled.

Files are stored in the **filetsf** spool directory in a hierarchy intended to make it easier to list and identify the files waiting to be retrieved. The structure is three levels deep below the spool directory, encoding the recipient, intended filename, and sender (user@host) in the directory names. The individual copies of files are named for the value of *time()* (the number of seconds since January 1, 1970) when the file was delivered to the spool directory by *tsfif*, with an optional prefix. This means that the pathnames under the spool directory are of the form

`recipient/name/sender/[prefix]time`

This structure allows multiple copies of the same file to be spooled from one or more senders for one or more recipients, providing a different pathname for each spooled file. Files are named for the time at which they were delivered using the integer returned by *time()* to allow for easy manipulation and sorting. Any name clashes discovered by *tsfif* can be resolved by waiting one second and regenerating the pathname.

Most special characters are prohibited in file, sender and recipient names, to avoid potential problems manipulating the files. We had considered making the spool hierarchy only one level deep, but we felt that that would have forced even more restrictions on the component names – we would have had to use some character or sequence of characters as a separator in the directory names.

A small number (currently three) of filename prefixes are used by **filetsf** to indicate the status of a particular version of a file. Temporary files, indicated by the # prefix, are used when writing a file into the spool directory, and are renamed to remove the prefix when they have been written correctly. A prefix of − is used to indicate a file that has been retrieved or deleted, but not yet purged from the system. And a prefix of + is used is to indicate that the newest version of the file should replace any older versions of the file still waiting to be retrieved. This is determined by the options used by the sender when initially sending the file, and can be useful for data such as a `hosts` file, or a telephone list, where only the most recent copy of the file is useful.

Files are transferred using **lpr/lpd** one file at a time, to keep things simple. The name (or desired name) of the file is sent as the LPD job name, and the LPD class is used to pass **filetsf** options and the name of the recipient. The options are encoded as an integer, and are separated from the recipient name by a colon.[4] The LPD protocol defines limits on the length of the control values that can be passed to the destination machine, but they haven't turned out to be unduly restrictive in practice.

Filetsf currently supports three options that are passed to the destination machine: notify the sender by email on delivery to the spool directory, notify the recipient on delivery, and mark the file as replacing any older versions of the file.

Command Descriptions

The core of **filetsf** consists of only three commands, *sendfile*, to send a file, *tsfif*, which is used as the **lpr/lpd** print filter and which puts files into the **filetsf** spool area, and *acceptfile*, to list or retrieve files from the spool area. In addition to the core commands, there are two utility programs, and a *tsfif* cover for use on AIX. The commands are written in C and the Bourne shell, and are about 2,000 lines of code in total.

The following is a short description of each **filetsf** command.

sendfile

Queues files for sending to a user on a remote machine. It provides options to provide a different name for the delivered file, to cause mail to be sent to the sender or the recipient when the file arrives, or to flag the file as replacing any previous copy of the file still in the spool directory. One the command line and files have been checked, *sendfile* invokes *lpr* on each file in turn to send the files to the destination machine.

acceptfile

Retrieves or lists files from the spool directory. It provides options to select spooled files by various attributes (sender, filename, newest, oldest, etc.), to rename a file on retrieval, and to delete files from the spool directory. *acceptfile* is setgid to the `filetsf` group, which allows it to manipulate the files in the spool directory, and is the only privileged program in **filetsf**.

tsfif

Print filter that writes transferred files into the spool directory. *tsfif* is invoked by the print system once for each file, and creates the appropriate directory structure and writes the file into the spool directory, taking

[4] We had hoped to use the title option to *lpr* for the recipient, and use class just for **filetsf** options, but we found that the title wasn't included in the control file passed to the remote machine, so that idea fell through.

care that the permissions and group of the files and directories are set as correctly as possible. Temporary files are used to insure that only complete and correct files are left in the spool directory for retrieval.

tsfif usually ends up being run by `daemon` (`lpd` on AIX), and the spool directory is owned by `daemon` (or `lpd`) and is in group `filetsf`. If BSD-style group inheritance is in not in use, `daemon` (or `lpd`) must be added to the `filetsf` group. If the group of the files and directories is or can be set to `filetsf`, then *tsfif* sets them group readable and writable; otherwise it leaves the permissions conservatively set, on the expectation that *ftcheck* will complain to a human about the configuration.

aixbe
: Print system backend program for use on AIX. The AIX printing system is IBM specific, and uses backend programs instead of print filters. *aixbe* does the appropriate things for AIX, and then calls *tsfif* to do the actual processing of a received file.

ftcleanup
: Cleans up the spool directory. Removes old temporary and deleted (but not purged) files, empty directories, and looks for old files that haven't been retrieved by the recipient yet and complains about them. Run daily by *cron*(8).

ftcheck
: Does a sanity check on the **filetsf** system. Checks for the `filetsf` group, checks the membership of the group, and makes sure that the spool directory and the files in it have the correct group.[5] It should really check the modes of the spool files and directories, but it doesn't (yet). Run daily by *cron*(8).

The VM/CMS client implementation was fairly simple; it's a REXX script called *filetsf* (since there's already a *sendfile* command on VM/CMS) that invokes the *netprint* command to send the file to the destination machine. Our VM/CMS system also has a command called *lpr*, but *netprint* serves the same function and is easier to use. Both *lpr* and *netprint* seem to be client-only implementations – they don't provide any spooling if the remote machine is down, which means that we have to manually retry file transfers from VM/CMS if the network or the destination system is down. We have not yet determined the best solution to this problem.

[5] *ftcheck* is interesting because it contains a one line *awk* program that happens to trigger a bug in *awk* on our copy of DEC OSF/1 V3.2A that causes *ftcheck* to complain incorrectly.

```
tsf-machine:\
     :mx#0:\
     :rp=filetsf:\
     :rm=machine:\
     :fx=f:\
     :lf=/var/lpr/tsf-machine/log:\
     :sd=/var/lpr/tsf-machine:
filetsf:\
     :lp=/dev/null:\
     :af=/dev/null:\
     :sd=/var/lpr/filetsf:\
     :lf=/var/lpr/filetsf/log:\
     :if=/local/filetsf/etc/tsfif:\
     :fx=f:
```

Figure 1: Sample `printcap` file entries

We hope that it will be possible to implement a **filetsf** server on VM/CMS, but have not yet investigated what it would entail. We're hoping that it will turn out to be a another simple REXX script and a little bit of configuration.

Installation and Configuration

Installation and configuration of **filetsf** is fairly straightforward. The programs require very little configuration to compile, and contain almost no machine-specific code, or non-standard library routines. The primary exception to this is, of course, *aixbe*. The programs can be installed just about anywhere in the filesystem.

There should be a sending queue named `tsf-machine` for each host `machine` that files will be sent to. Each destination machine needs a queue called `filetsf` with *tsfif* as the print filter. Sample entries are shown in figure 1. The *filetsf-config*(7) man page gives (hopefully) complete details on userids, groups, `printcap` file entries, and queue configuration for both **lpr/lpd** based systems and AIX.

Examples of Use

Filetsf is intended to be easy to use. This is an very important feature for us, since we wish to convince the application programmers in our department to use this new file transfer system, rather than old familiar FTP. It's hard to convince anyone to use something that's more complicated than what they're already used to.

The simplest example of sending a file is

```
% sendfile -h otherhost file
```

which sends `file` to the same user on host `otherhost`.

To distribute a file like `/etc/hosts` to a different machine, the command would be

```
% sendfile -h otherhost \
  -r -u root /etc/hosts
```

The *-r* option indicates that this copy should replace any copy still in the **filetsf** spool directory on `otherhost`, and the *-u* option indicates that the file should be sent to `root`, regardless of who is sending the file (presumably `root` is the only user with permission to update the `/etc/hosts` file). The *sendfile* command only allows one *-h* option and one *-u* option, so multiple invocations of *sendfile* are required to send a file to more than one recipient. Multiple files can be sent by one *sendfile* command, so, for example, the `/etc/networks` file could be sent at the same time as the `/etc/hosts` file.

The *acceptfile* command is a little more complicated to use, since it may be necessary to select a particular version of a file. The *-l* option to *acceptfile* generates a list of the queued files, such as that shown in figure 2. The output is the recipient userid, file name, sender and the instance of the file, with any repeated information suppressed. The `r` preceding three of the file instances in the example indicates that those files were sent with the *-r* option to *sendfile*. The *-c* option to *acceptfile* can be used with *-l* to list the files in columns, for easier parsing by other programs.

To retrieve a file, *acceptfile* must be invoked with options that result in the selection of exactly one file. The selection options can also be used with *-l* in order to test the selection options. For example,

```
% acceptfile -n hosts
```

would retrieve the newest version of the file `hosts` into the current directory. The *-o* option can be used to select the oldest version of a file, so that files can be processed in the order they were sent. A file labeled with a particular time can be selected with the *-t* option, as in

```
% acceptfile -t 805922175 hosts
```

Files can be selected by sender with the *-s* option, as in

```
% acceptfile -s \
  jms@mach2.uwaterloo.ca hosts
```

The retrieved file can be renamed or placed in a different directory with the *-r* option. Files can be deleted from the spool directory without being retrieved by using the *-d* option to *acceptfile*. When a file is retrieved or deleted, *acceptfile* renames the spooled copy of the file for later removal by *ftcleanup*. The *-p* (purge) option can be used to cause the files to be removed from the spool directory immediately.

Security and Reliability

As with any other business system, a file transfer system needs to be reviewed to ensure that it provides an appropriate level of security and reliability for the applications in which it will be used. We will discuss the security and reliability aspects of **filetsf** in a step by step review of the file transfer process.

In our environment, there are no new security implications in the act of sending a file. *sendfile* is not privileged, so the sender of a file must be able to read the file already. Since email and FTP are available on our systems, *sendfile* does not provide any confidentiality exposures that don't already exist. In the interest of reliability, *sendfile* does not send any files until it has tested all the files for existence and readability, so that the sending of a set of files is less likely to fail part way through.

The **lpr/lpd** system is reasonably secure and reasonably reliable – we have a reasonable expectation that we can control which machines have access to our print queues and the data flowing through them. Our administrative systems are on subnets separate from the rest of the campus, our routers can block LPD protocol traffic (if necessary), and we use a locally modified **lpr/lpd** (a precursor to PLP[6]) that allows us to implement per-queue access controls, so that we can limit the machines that we are willing to trust to exchange files with. This allows us to trust the information in an LPD control file once it reaches the destination machine, so that we can label a file as being sent by a particular sender with a reasonable level of confidence. And **lpr/lpd** tries to deal correctly with full print spool directories and network interruptions, which allows us to believe that a file that has been passed to **lpr/lpd** will eventually show up in the right place, without being corrupted along the way.

tsfif is run on the destination machine by *lpd*, usually as the `daemon` user, and creates sub-directories in the spool

[6]The most recent version of PLP is available from `ftp.iona.ie` in the `pub/plp` directory. See also "LPRng – An Enhanced Printer Spooler System" by Patrick Powell, elsewhere in these proceedings.

```
% acceptfile -l
jms
    hosts
        jms@mach1.uwaterloo.ca
            805922115 Sun Jul 16 15:15:15 1995
            805926809 Sun Jul 16 16:33:29 1995
        jms@mach2.uwaterloo.ca
           r805922175 Sun Jul 16 15:16:15 1995
    networks
        jms@mach2.uwaterloo.ca
           r805922175 Sun Jul 16 15:16:15 1995
           r805923542 Sun Jul 16 15:39:02 1995
% acceptfile -l -c
jms hosts jms@mach1.uwaterloo.ca  805922115 Sun Jul 16 15:15:15 1995
jms hosts jms@mach1.uwaterloo.ca  805926809 Sun Jul 16 16:33:29 1995
jms hosts jms@mach2.uwaterloo.ca r805922175 Sun Jul 16 15:16:15 1995
jms networks jms@mach2.uwaterloo.ca r805922175 Sun Jul 16 15:16:15 1995
jms networks jms@mach2.uwaterloo.ca r805923542 Sun Jul 16 15:39:02 1995
```

Figure 2: Sample *acceptfile* output

directory and writes the transferred files into the appropriate directory. The primary concern with *tsfif* is that it needs to be careful that it does not deposit a file in the spool directory over an existing file or symbolic link. This risk is minimal, since the holding directory has no general permissions on it, making it unlikely that someone could create a dangling symbolic link in the spool directory. If *tsfif* runs into any problems, such as permission or ownership errors, or a full spool directory, it either writes the file with restrictive permissions, or fails with an exit code that causes *lpd* to keep trying to deliver the file. This means that, for some kinds of errors, the `filetsf` print queue will stop delivering files. We chose to keep retrying because the only alternative was to give up on a file, and have that particular transfer get lost, and we felt it was very important to protect the files being transferred.

acceptfile is a privileged program; it is setgid to the `filetsf` group. It is very careful about what it does since it has privileges that an ordinary user does not have, and it allows a user to make use of those privileges. *acceptfile* is very careful about identifying its invoker, and it renounces its privileges before writing a file to its final destination, so that a user should not be able to change a file with *acceptfile* that can't already be changed by the user. For reliability, *acceptfile* renames files in the spool directory, rather than deleting them after delivery. If something goes wrong in later processing, and the delivered file is lost or destroyed, there may be a copy that could be recovered from the **filetsf** system (by a super-user).

The **filetsf** system, through the use of the `filetsf` group, tries to limit its potential exposures. If the `filetsf` group is somehow compromised, there should be limited exposure to the rest of the machine, since nothing else should be using the `filetsf` group.

Filetsf isn't 100% secure or 100% reliable; as in any system, the risks have to be weighed against the potential consequences and the costs to eliminate the risks. The **filetsf** system provides, for us, an acceptable level of security and reliability.

Potential Extensions

Filetsf was planned to be a simple file transfer system, providing the necessary, basic functions, and providing appropriate building blocks on which to implement more enhanced functionality (we'll claim that this is the "UNIX philosophy" again). While **filetsf** seems to work just as it is, there are a number of potential extensions to **filetsf** that may be worth investigating.

Some file transfers will likely require greater security than that provided by **filetsf**, due to greater potential exposures. Since **filetsf** is just sending files, file encryption, acknowledgement of receipt, checksums, etc., could be built on top of **filetsf**.

We sometimes have the need to send sets of files, for processing as a unit. The simple approach to handling sets of files is to combine the separate files into a single file, using *tar*(1), or some other file archiving program. An alternative that might be useful would be to create a *getset* command, that would check to make sure that the complete set of files are available, retrieve them all, and run a command to process the files. This is something that is simple enough to do, and it may be a common enough operation to justify creating a command to do it.

There may be situations in which the **lpr/lpd** system or the LPD protocol is either not appropriate or not available. For example, there might be a need to transfer files over larger, public networks in a secure fashion, or through networks that do not allow LPD protocol traffic. It should be fairly easy to implement **filetsf** on top of a different method of transport, or to use multiple transports depending on the sender and recipient.

Conclusions

Filetsf seems to meet our needs for a basic file transfer system, and is an improvement over our past approaches to file transfer. While not providing extreme levels of security, it is a useful tool on our systems, and we expect to make fairly extensive use of it in the future. It was relatively easy to implement, and, if our needs change in the future, it should be a fairly straightforward task to replace **lpr/lpd** with some other transport mechanism providing greater reliability or security, while leaving the user interface unchanged.

Acknowledgements

The VM/CMS client implementation is due to the efforts and knowledge of Cameron McDonald.

Author Information

John Sellens is a Chartered Accountant and holds a master's degree in Computer Science from the University of Waterloo. He is currently Project Leader, Technical Services in the Data Processing department at the University of Waterloo where he is responsible for system administration and planning, and tool development for the administrative computing systems. John likes to provide as long a list of references as possible in each paper that he writes, whether or not he has actually read whatever it is he's referring to. John can be reached by mail at Data Processing, University of Waterloo, Waterloo, ON N2L 3G1, Canada, or electronically as `jmsellens@uwaterloo.ca`.

Availability

The current version of **filetsf** is available through anonymous FTP to `math.uwaterloo.ca` as `pub/filetsf/filetsf.tar.Z` or through the author. Installation and configuration should be fairly straightforward at most sites. Like most freely available software, no support is available for **filetsf**, but bug fixes and suggested enhancements would be gratefully accepted.

References

[1] Anderson, Paul, "Managing Program Binaries In a Heterogeneous UNIX Network", *LISA V Proceedings*, San Diego, CA, October, 1991, pp. 1-9.

[2] Borenstein, N., and N. Freed, *MIME (Multipurpose Internet Mail Extensions): Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, RFC 1521, September, 1993.

[3] Campbell, Ralph, *4.3BSD Line Printer Spooler Manual*, Computer Systems Research Group, University of California, Berkeley, 1986.

[4] Colyer, Wallace, and Walter Wong, "Depot: A Tool for Managing Software Environments", *LISA VI Proceedings*, Long Beach, CA, October, 1992, pp. 153-159.

[5] Cooper, Michael, "Overhauling Rdist for the '90s", *LISA VI Proceedings*, Long Beach, CA, October, 1992, pp. 175-188.

[6] Dagenais, Michel, et al, "LUDE: A Distributed Software Library", *LISA VII Proceedings*, Monterey, CA, November, 1993, pp. 25-32.

[7] Manheimer, K., B. A. Warsaw, S. N. Clark, and W. Rowe, "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries", *LISA IV Proceedings*, Colorado Springs, CO, October, 1990, pp. 37-46.

[8] McLaughlin III, L., ed., *Line Printer Daemon Protocol*, RFC 1179, August, 1990.

[9] Nachbar, Daniel, "When Network File Systems Aren't Enough: Automatic Software Distribution Revisited.", *Proceedings of the Summer USENIX Conference*, Atlanta, GA, June, 1986, pp. 159-171.

[10] Postel, Jonathan B., *Simple Mail Transfer Protocol*, RFC 821, August, 1982.

[11] Postel, J., and J. Reynolds, *File Transfer Protocol*, RFC 959, October, 1985.

[12] Sellens, John, "Software Maintenance in a Campus Environment: The Xhier Approach", *LISA V Proceedings*, San Diego, CA, October, 1991, pp. 21-28.

[13] Wong, Walter C., "Local Disk Depot – Customizing the Software Environment", *LISA VII Proceedings*, Monterey, CA, November, 1993, pp. 51-55.

[14] Zimmerman, Philip, *The Official PGP User's Guide*, ISBN 0-262-74017-6, MIT Press, 1995.