Proceedings IEEE INFOCOM '83, San Diego, CA, pp. 168-175, Apr. 1983

STORK: AN EXPERIMENTAL MIGRATING FILE SYSTEM FOR COMPUTER NETWORKS

Jehan-François Pâris

Department of EE & CS, University of California, San Diego
La Jolla , CA 92093

Walter F. Tichy

Department of Computer Sciences, Purdue University
West Lafayette, IN 47907

Abstract

STORK is an experimental file system designed for local and long-haul networks. It ensures that each user has a single view of his files, independent of the network node where he works, and independent of the location of the files. STORK files have no fixed location; instead they migrate to the network node where they are needed. File consistency is ensured by permitting only one current copy of each file to exist in the net at any given time. A lock mechanism is provided for controlling concurrent access.

The performance of the system depends on the locality of the references to a given file and not on the host where the file was created. An analytical model is presented that compares file migration with remote file access.

STORK has been implemented on a network of UNIX* systems running on VAXes and PDP-11's, using primitives of the Berkeley Network software. It can also be quickly installed on any network of UNIX systems allowing remote execution of commands.

Keywords: networks; distributed file systems; file transfer; migration; UNIX operating system.

1. INTRODUCTION

Modern operating systems implement abstract machines that are more convenient to use than bare hardware is. They hide low-level details of managing hardware and software resources, and present powerful virtual machines for user programs. For example, the locations of files on a disk are irrelevant -- the operating system performs the mapping of file names to disk addresses automatically.

Network services are a relatively recent addition to operating systems. A wide variety of communication technologies is available, providing reliable, low cost, high-speed computer communication. Although the underlying hardware and method of transmission vary significantly from network to network, layers of controlling software hide these differences. Thus it is possible to provide a unique application-level interface that works over a variety of configurations.

The services offered by current networks vary widely with the speed of the links available, the integration of the different systems, and the sophistication of the network software. File transfer, message transfer (electronic mail), and the possibility of executing commands on remote hosts are examples of basic capabilities provided by nearly all networks. More advanced networks also support services like remote login, remote interactive commands [Hwan82], or operations on files residing on remote hosts [Rowe82].

These networks share a common drawback: they require the user to remember where he is currently logged in and where his files are. This property complicates the user's task considerably, because it violates the principle of location transparency. Location transparency permits a user to reference files by name rather than by address. Most centralized file systems implement location transparency by mapping file names to disk addresses automatically. Current networks add an additional level to the naming hierarchy, but provide no automatic location transparency. Thus, in using networks, the user must take a step backwards and deal with files at a lower level of abstraction than with files on traditional operating systems.

What the user really needs is a true network file system spanning the secondary stores of all network hosts. Such a system is able to provide users at all sites with the same view of the file system,

*UNIX is a trademark of Western Electric

independent of where files actually reside.

The STORK system described in this paper is an experimental file system that provides complete location transparency for files scattered throughout a network. It is implemented on a net of UNIX machines using the Berkeley Network software. STORK is based upon the idea of file migration. This means that files do not have a fixed home; instead, they migrate to network nodes where needed. For example, a programmer may create a file on one machine, but edit it on another. Referencing the file to be edited on the second machine is sufficient to force the file system to transfer the file to where it will be needed. This approach assures fast access if the file reference patterns exhibit locality at the host level.

An important assumption is that users tend to access a certain set of files repeatedly from the same host, giving migrating files a considerable performance advantage over files with fixed homes. STORK has been built in part to test whether this assumption is justified, and to measure the effects of migration.

A major constraint in building STORK was that we could not afford to redesign the file system of every host on the network. Instead, we took the approach of extending an existing file system, the Unix file system, which is supported by most hosts on our network. To simplify integration, STORK provides an interface similar to that of the Unix file system, with only minor extensions.

Section 2 of this paper summarizes the approaches to file handling on networks. Section 3 presents the STORK system, and Section 4 gives an analytical performance comparison of STORK with remote file operations. Section 5 discusses our current implementation based on the UNIX operating system and the Berkeley Network software. Our conclusions appear in Section 6.

2. ACCESS TO REMOTE FILES

There are three alternative approaches to accessing remote files: remote file operations, remote login, and file migration. As we shall see, only remote file operations and file migration are suitable for supporting a network file system.

2.1 Remote File Operations

Remote file operations allow the opening, reading, writing, and closing of remote files through the network. At the application level, local and remote files are treated in exactly the same way. COCANET is a typical example of this approach [Rowe82]. It is implemented on a net of computers running the UNIX operating system, and allows programs to access remote resources, in particular remote files. COCANET requires that the names of all remote files be prefixed with the name of the host where the file is stored. When a process attempts to open a file whose name is prefixed by the name of a non-local host, control is passed from the standard UNIX open routine to the network manager. The network manager communicates with its counterpart in the remote host to create a private remote server for the process attempting the open. All further operations on the remote file are passed to the remote server, which executes them and transfers the results back through the net.

The main drawback of COCANET is that it requires the user to remember where his files are located. There is no attempt to provide location transparency on the host level.

A potential advantage of remote file operations is that they send only those buffers that are actually read or written through the net. Thus, remote file operations handle accesses to relatively small portions of large files efficiently. Such a situation is often found in data base applications. Remote file operations are somewhat less efficient if files are accessed completely and repeatedly from a single host. This is typically the case in program development or text processing applications that account for a large part of the workload of many interactive systems. For example, UNIX text editors do not normally overwrite portions of a file -- when the editing session is finished, they write a completely new file and delete the old one. Moreover, text processing and program development typically require many iterations and therefore repeated access from the same host to the same set of files. A file system based on remote file operations would move the whole content of each file back and forth between the two hosts every time the file is accessed. These transfers not only take time, but may also be so numerous as to exceed the capacity of the network.

2.2. Remote Login

Another approach to the problem of

accessing remote files is remote login. This facility allows any terminal connected to the local host to act as if it were directly wired to a remote host. Such a facility is provided by many networks, for instance ARPANET, CSNET, Telenet, etc. However, remote login has a serious restriction: at any time, the user can access conveniently only those files residing at the machine where he is logged in. If he wants to work with files at other machines, he must wander around in the net to get to them. If he needs to combine several files, he must transfer them to a single host. This is a rather severe limitation which precludes the use of remote login for supporting a network file system.

2.3. File Migration

The concept of file migration is not new. It has been implemented on several computer installations and studied intensively, for example by Stritter [Strit77], Smith [Smit81a, Smit81b], Lawrie e.a. [Lawr82], and Porcar [Porc82]. With one exception, previous research has only considered file migration within the memory hierarchy of a single computer. The exception is Porcar's thesis, which analyzes several strategies for a distributed file system, including migrating files. His study relies on data obtained on a single machine.

Supporting migration requires few changes to host file systems. Once a file has been migrated to a given host, it can be accessed there using the file primitives of the host operating system. Concurrency control is simple, since the current copy of the file is always local to the host updating it. Migration performs best when files are repeatedly referenced from the same host rather than accessed at random. It is our contention that this behavior occurs frequently enough to make file migration more efficient than remote file operations.

3. THE STORK SYSTEM.

Since a true network file system completely hides the location of files from its users, the system can assume all responsibility for file placement. As a consequence, files do not need fixed homes. Instead, they are free to migrate to network hosts where needed or where there is space available for them. For example, a programmer may work on several different hosts. Referencing a file on any host is sufficient to force the network to locate the file and bring it to that host. In addition, files that are located on hosts where file space is at premium may be transferred automatically to less congested hosts.

3.1. The File Search Mechanism

On a broadcast network, name searches can be managed with a single broadcast message. On networks without any broadcast mechanism, some directory information must be kept to locate files efficiently. In STORK, this information is minimized by providing each user with a host search list, consisting of a number of hosts to be interrogated when locating files. Whenever a file cannot be found locally, STORK sends a request for the file to every host on the host search list until one responds by sending the file to the requesting host. Now suppose that we had a hybrid network consisting of broadcast (sub)nets linked by gateways. The host search list would then be a list of gateways; each of these gateways would broadcast the request on its respective subnet.

As a consequence, the name of a file and the directory where it resides must be the same for all hosts included in the system. This restriction is necessary to assure uniform file names on all hosts. The local directory hierarchy of every host is simply an image, sometimes incomplete, of the directory hierarchy of the network file system.

A common way of organizing a file system is to let each user manage his own directory tree. Unfortunately, administrators of computing facilities have differing opinions of what the root names of these directory trees should be. For example, John Doe could have the root directory names doe, john, jdoe, jxd, or even a random string on the various network hosts. For such cases, STORK provides a simple root name translation feature.

3.2. Concurrency Control

Like any file system on a single machine, the distributed file system must contend with the problem of concurrent access. The design of STORK aims for simplicity and utility, rather than generality. The system permits multiple simultaneous readers of a file, but only a single writer. Readers may obtain a copy of the file when they first access it; updates written to a file while it is being read are not guaranteed to be visible to the readers. The system also provides locking primitives for ensuring exclusive access to a file.

3.3. File Access Rights

An important requirement of a distributed file system is a file protection mechanism similar to those existing in centralized file systems. STORK satisfies this requirement simply by preserving the ownership and protection attributes of a migrated file.

A somewhat more complicated issue involves the right to migrate a file to a different node in the network. The owner of a file may not wish to permit others to move his file away, because he will then incur the delay of transmitting the file back. One solution is to introduce a file migration right, analogous to the well-known read/write rights for files. Thus, the owner of a file can decide what is most appropriate, independent of other access rights. This approach has two main drawbacks. First, it makes the users aware of the migration process and therefore contradicts our objective of location transparency. Second, it raises the problem of how to access remote files that cannot be migrated. A process requesting read-only access is easily taken care of by providing it with a temporary local copy of the file. A process that wants to write a non-migratable file causes more difficulties. One approach would be to perform the write by means of remote file operations similar to COCANET. Another one would be to make a local copy of the file, to let the process update that copy, and to update the remote original when the process closes the file.

A simpler solution to the migrate right problem is to associate the migrate right with the write right. This technique could inconvenience the owner of a file, since he might have to retrieve it after it has been migrated to other nodes. On the other hand, this solution is fair in that it does not give preference to a particular writer, and it is consistent with location transparency.

3.4. STORK File Access Primitives

STORK's aim is to hide the location of files. Each time a process reads or writes a file, STORK tries to seize it, i. e. to cache it in the host on which the process is running. The precise semantics of the seize operation are as follows. If the file already resides on the local machine, seize checks for a lock and returns successfully if the file is not locked. If no local copy of the file is found, seize successively interrogates all hosts in the host search list. If the file is found, seize determines whether it is locked or unlocked and whether the process initiating the seize has the permission to migrate the file.

If the file is unlocked and the requestor has migrate permission, seize creates a local copy of the file, under the same name and with the same same access rights as the remote copy of the file. Seize also marks the remote copy as stale. In all other cases, seize returns an error code describing the status of the file (not found, locked, or migrate permission denied).

The primary purpose of the latch and unlatch primitives is to ensure concurrency control by prohibiting concurrent writes of the same file. Latch creates a lock for a file, preventing other processes from writing and migrating the file. Latch fails if the file is not local, so it must normally be preceded by a call to seize. Unlatch removes the lock.

The primitive operation carbon-copy provides read-only access to files which are locked or cannot be migrated. Carbon-copy creates an input stream from which the requesting process can read the contents of the file. If the file is non-local, carbon-copy first copies it into a a local file that remains inaccessible to the caller. (Users that want to avoid the delays resulting from repeated carbon-copy operations on the same file can always make their own local copy of the file.)

Seize and carbon-copy are both low-level operations. The user interface of STORK consists of network analogues of the usual file primitives open, create, close, etc. For instance, net open is the analogue to the open primitive and is implemented as follows:

```
procedure net_open(file, mode)
begin
    if seize(file)=successful then
        open(file, mode)
    elsif mode=read-only then
        carbon-copy(file,tempfile)
        open (tempfile, read)
    else
        fail
    end if
end net_open;
```

4. PERFORMANCE ANALYSIS

Let us neglect for a moment the problem of concurrent accesses and assume that files are always seized and never carbon-copied. Under this assumption, STORK files always reside on the host where they have been accessed most recently. Since users

generally tend to access the same file from the same host repeatedly, the file system continuously adapts itself to the demands of its users. This flexibility has a price. Since files have no fixed addresses, the search for a non-local file normally involves the interrogation of several remote hosts.

Consider a computer network consisting of n hosts numbered 1, 2, ..., n. Assume that the status of a file can be modeled by a first-order Markov chain whose states correspond to the nodes where the file was accessed for the last time -- and thus where it resides. The transition probability matrix of the chain can then be written as $P = (p_{ij})$, where p_{ij} is the probability that the file is accessed from host j, given that it was accessed for the last time at host i. Obviously, the sum of these probabilities must be one.

$$\sum_{j=1}^{m} p_{ij}=1, \qquad i=1, 2, \ldots, n.$$

Assuming that the chain is homogeneous, irreducible and aperiodic, one can compute its limiting state probability vector as $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n)$. This vector λ is the eigenvector of matrix P. Each λ_i represents both

-- the steady-state probability that the file will be accessed next from host i, and

-- the steady-state probability that the file resides on host i.

This first-order chain can simulate the behavior of distributed files with different degrees of locality . For the sake of simplicity, we restrict ourselves to a special first-order Markov model that only considers consecutive references to the same file from the same host. This model, known as Easton's model, was originally developed for modeling the page referencing behavior of data bases [East75]. It requires n+1 rather than n^2 parameters, but can nevertheless take into account the tendency of files to be repeatedly accessed from the same host. The transition probabilities of Easton's model are given by

$$p_{ii} = r+(1-r)\lambda_i$$

$$p_{ij} = (1-r)\lambda_j, \qquad i \neq j,$$

where $0<r<1$ and $\lambda_i>0$ for $i = 1, \ldots, n$.

Since

$$\sum_{j=1}^{n} p_{ij} = 1, \qquad i=1,\ldots,n,$$

one must necessarily have

$$\sum_{i=1}^{n} \lambda_i=1.$$

To evaluate the cost of seizing a remote file on STORK, one must consider both the number of remote hosts interrogated during a seize and the number of blocks shipped over the network.

The average number of hosts interrogated when seizing a file depends both on the probability of finding the file on any given host and on the order of the host search list. To stay on the conservative side, we assume that the ordering of hosts in the search list has been fixed arbitrarily and was not tuned to the individual behavior of any file. We also assume that the probability of searching for a non-existing file is negligible. Under this assumption, the average number of hosts to be interrogated for locating a file not found on the local host is $\frac{n-1}{2}$.

At any given time, there is a probability λ_i that the file resides on host i. The probability that it will be accessed from host j while being away from that host is equal to

$$\sum_{i\neq j} \lambda_i p_{ij} = (1-\lambda_j)(1-r)\lambda_j.$$

The average number $\bar{N}_m$ of hosts to be interrogated when a non-local file is seized is then given by

$$\bar{N}_m = \sum_{j=1}^{n} (1-\lambda_j)(1-r)\lambda_j \frac{n-1}{2}, \qquad (1)$$

and the average number $\bar{T}_m$ of blocks to be shipped is

$$\bar{T}_m = \sum_{j=1}^{n} (1-\lambda_j)(1-r)\lambda_j S, \qquad (2)$$

where S is the size of the file in blocks.

Suppose now that the same file is accessed via remote file operations as provided by COCANET. To be fair, let us assume that the file resides at the host where it is the most frequently referenced. (A more detailed discussion of the problem of optimal file placement can be found in [Chu69].) The average number $\bar{N}_r$ of remote requests when the file is accessed is now equal to

$$\bar{N}_r = 1-\lambda_{max}, \qquad (3)$$

where $\lambda_{max} = \max [\lambda_i | i=1, \ldots, n]$. The average number $\bar{T}_m$ of blocks shipped is equal to

$$T_r = (1-\lambda_{max})(R+W), \qquad (4)$$

where R is the number of blocks read and W the number of blocks written.

Observe now that

$$\sum_{j=1}^{n} (1-\lambda_j)\lambda_j \leq (1-\lambda_{max})\lambda_{max} + (1-\lambda_{max}) \leq (1-\lambda_{max})(1+\lambda_{max}).$$

One has then

$$(1-r)(1-\lambda_{max})(n-1)/2 \leq \overline{N}_m \leq (1-r)(1-\lambda_{max})(1+\lambda_{max})(n-1)/2$$

and

$$(1-r)(1-\lambda_{max})S \leq \overline{T}_m \leq (1-r)(1-\lambda_{max})(1+\lambda_{max})S,$$

where S is the size of the file in blocks.

Comparing these two last inequalities with equations (3) and (4), one can see that migrating file systems constitute the policy of choice when files are accessed with a high degree of locality ($r \simeq 1$). On the other hand, network file systems perform better when only a small portion of the file is needed at each access ($R+W<<S$).

5. THE STORK IMPLEMENTATION

Our main objective was to build a prototype of a migrating file system that was quickly implemented and easily instrumented. The fastest way to implement STORK was to use the command language C-shell [Joy80] as the programming language and to use an existing network, the Berkeley Network, for communication between hosts. Since transmission delays are the major influence on the performance of the new commands, the performance penalty stemming from an interpretive command language was judged insignificant.

The Berkeley Network [Schm80] provides facilities for file transfer, sending and receiving mail, and remote printing. It was designed for low cost and operates in a batch mode, not unlike a line printer queue. At its lowest level, the original Berkeley Network transmits data over TTY lines through terminal interfaces and system drivers acting as if characters were coming from terminals. This solution was adopted for its low implementation cost, but has the disadvantage of permitting only extremely slow transmission rates (between 1200 and 9600 baud).

Thanks to a local modification of the network software, the Berkeley Network at Purdue University now runs on top of the ARPANET protocol TCP/IP [Post81], which allows it to use the local PRONET ring connecting the two VAXes of the Computer Science Department. The PRONET hardware achieves transmission rates of up to 1 Mbaud and alleviates the bottleneck caused by the slow TTY lines. A third machine, a PDP-11/70, is also connected, but only via 9600 baud lines.

5.1. File Handling

In STORK, the single current copy of a file resides on the network host where the file was successfully seized last. In order to keep the system simple to use, the current copy of any STORK file is represented as a standard Unix file, and may be accessed without additional overhead at the current host. One may seize a standard UNIX file existing on a remote host as long as one has the proper migrate permission. There are no special data structures or additional parameters needed in order to make a file migrate.

Keeping a single copy of every file has the drawback that the file may be lost if the host to which it migrated is unreliable. This problem is alleviated in STORK by not deleting the file when it migrates away. Instead, the seize operation merely renames the original file, making it it invisible to the user. Thus, a stale copy of the file exists on every host visited by the file. In case of a loss, the most recent stale version of the file can be recovered. Stale copies are deleted automatically if they have not been accessed in three consecutive days. Before deletion, they are saved on backup tape.

5.2. The STORK Primitives

A full implementation of a network file system under UNIX would require modifying all UNIX programs to perform seize or carbon-copy operations on all non-temporary files. Such a task was clearly beyond our goal of building a prototype. Instead, we decided to put the burden of conversion on the users, providing them only with a set of STORK primitives, some to be included in shell scripts, others in C programs.

The STORK primitives that may be included in shell scripts are as follows:

Seize migrates a file to the local host and is defined as follows.

```
procedure seize(file)
if file is local then
    if file is locked or file in transfer
    then fail
    end if
else
    forall hosts in sitepath do
        if file is present then
            if file is locked
                or file in transfer
                or no migrate permission
            then
                fail
            else
                set in_transfer flag
                mark remote file stale
                ship file to local host
                reset in_transfer flag
                return success
            end if
        end if
    end forall
    fail --file not found
end
```

Carbon-copy lists the content of a file on the standard output of the local host.

Latch locks a (local) file.

Unlatch unlocks a (local) file.

Net ls performs a Unix ls operation on all machines given by the variable sitepath and coalesces the results.

These five primitives are implemented as shell programs and expect the STORK user to have a shell variable sitepath containing the host search list. The migrate right is implemented with the read/write rights of the directory containing the file. (Thus, migrate rights are handled at the directory level and not at the file level as they should be.) STORK also implements the primitive net open, discussed earlier, as a C-program.

As its present stage, STORK is as yet far from providing all of the services one expects from a true network file system. For instance, it does not contain any mechanism for avoiding name conflicts between files residing on different hosts and always requires the explicit use of STORK primitives to access non-local files. A main reason for these limitations lies in the slowness of the network. On heavily loaded hosts, the transfer time for a file may rise from the normal 10-30 seconds to 1-3 minutes. The reason for this poor performance lies in our reliance on the Berkeley Network software for handling all communications between hosts. The Berkeley Network is essentially a batch-oriented system, not aimed at providing fast turn-around for short requests. Version 4.1a of Berkeley Unix includes more efficient protocols for inter-host communication. STORK is being currently rewritten at the University of California, San Diego using these protocols and we expect dramatic improvements of the response times of our system.

5.3. Directions for Future Work

A main purpose of the STORK system is to demonstrate the feasibility of file migration as an implementation technique for network file systems, and to allow the collection of data on file access patterns in distributed environments. Such data are an absolute prerequisite for better understanding distributed file systems and the design of better file migration algorithms. We plan to insert data-gathering facilities into the STORK primitives that monitor, among other things, how often files are accessed and when they migrate.

A more practical problem with the current implementation of STORK lies in its relative slowness. We plan to explore the use of better protocols, and possibly reimplement STORK using TCP/IP.

A host of other issues remains. A more efficient mechanism should replace the current sitepath solution. The system should detect name conflicts occurring when two files residing on different nodes are given the same name. The migrate right should be handled better. Multiple copies of files should be permitted, at least for read-only files. This raises the problem of updating multiple copies when they turn stale. Version control is another issue. In software development, it is often useful to save multiple versions of software, in order to have back-up copies and to be able to maintain old versions [Tic82]. Providing redundancy and version control will require a complete redesign of STORK.

6. CONCLUSIONS

We have presented an experimental network file system aimed at local and long-haul networks. It provides users at all sites with a uniform and location-independent method for accessing files. Files do not have a fixed home, but migrate within the network to the host where needed. Such a scheme is especially well tailored to situations where files are repeatedly accessed from one host before being requested from another host. Our system ensures file consistency by allowing only one active copy of a given file at any time. It provides a lock mechanism for controlling concurrent

access to a file.

STORK has been implemented on a network of UNIX systems using the Berkeley Network software and standard UNIX tools. As a result, our software is compact, highly portable and can be quickly installed on any standard UNIX system supporting the Berkeley Network. Since our design is by no means specific to UNIX systems, it could be applied to other computer networks providing an efficient file transfer protocol and remote execution of commands.

Acknowledgements

The work reported here was supported in part by NSF grant MCS-8109513. We would like to thank Professor P. J. Denning, Purdue University, for his comments and suggestions.

References

[Bour78] Bourne, S. R., "The UNIX Shell," The Bell System Technical J. 57, 6 Part 2 (Jul.-Aug. 1978), 1971-1990.

[Chu69] Chu, W. W., "Optimal File Allocation in A Multiple Computer System," IEEE Trans. Comput., Vol .C-18, No. 10 (Oct. 1969), 885-889.

[East75] Easton, M. C., "Model for Interactive Data Base Reference Strings," IBM J. Res. Develop., 19, (Nov. 75), 550-556.

[Hwan82] Hwang, K., W. J. Croft, G. H. Goble, B. W. Wah, F. A. Briggs, W. R. Simmons, and C. L. Coates, "A UNIX-Based Local Computer Network with Load Balancing," Computer, Vol. 15, No. 4 (April 1982), 55-65.

[Joy80] Joy, W., "An Introduction to the C Shell," Fourth Berkeley Software Distribution UNIX Documentation, Department of EECS, University of California, Berkeley, Calif., 1980.

[Kern81] Kernighan, B. W. and J. R. Mashey, "The UNIX Programming Environment," Computer 14, 4 (April 1981), 12-24.

[Lawr82] Lawrie, D. H., J. M. Randal, and R. R. Barton "Experiments with Automatic File Migration," Computer, Vol. 15, No. 7 (July 1982), 45-56.

[Porc82] Porcar, J. "File Migration in Distributed Computer Systems", Ph. D. Dissertation, Department of EECS, University of California, Berkeley, Calif., 1982. (also available as Report LBL-14763, Lawrence Berkeley Laboratories, Berkeley, Calif., 1982.)

[Post81] Postel, J. "DARPA Internet Program Protocol Specifications," RFC's 790-796, USC Information Sciences Institute, Marina del Rey, Calif., 1981.

[Ritc74] Ritchie, D. M. and K. L. Thompson, "The UNIX Time-Sharing System," Comm. ACM 17, 7 (Jul. 1974), 365-375. A revised version appeared in The Bell System Technical J. 57, 6 part 2 (Jul.-Aug. 1978), 1295-1990.

[Ritc78] Ritchie, D. M., S. C. Johnson, M.E. Lesk and B. W. Kernighan, "The C Programming Language," The Bell System Technical J. 57, 6 Part 2 (Jul.-Aug. 1978), 1991-2019.

[Rowe82] Rowe, L. A. and K. P. Birman, "A Local Network Based on the UNIX Operating System," IEEE Trans. Software Engineering, Vol. SE-8, No. 2 (Mar. 1982), 137-146.

[Schm80] Schmidt, E., "An Introduction to the Berkeley Network," Fourth Berkeley Software Distribution UNIX Documentation, Department of EECS, University of California, Berkeley, Calif., 1980.

[Smit81a] Smith, A. J. "Analysis of Long-Term File Migration Patterns," IEEE Trans. Software Engineering, Vol. SE-7, No. 4, 403-417.

[Smit81b] Smith, A. J. "Long-Term File Migration: Development and Analysis of Algorithms," Comm. ACM, Vol. 24, No. 8 (Aug. 1981), 521-532.

[Strit77] Stritter, E. P., "File Migration," Ph. D. Dissertation, Stanford Univ. Computer Sci. Rep. STAN-CS-77-594, Jan. 1977.

[Tic82] Tichy, W. F. "Design, Implementation, and Evaluation of a Revision Control System," Proceedings of the 6th International Conference on Software Engineering, Tokyo, Sept. 1982.