A Workload Model for Large Distributed File Systems

Srinivasan Keshav
David P. Anderson

Computer Science Division,
Department of EECS,
University of California, Berkeley.

May 6, 1988

ABSTRACT

We are developing a simulation-based system for design evaluation of very large distributed file services. This paper describes the workload generation component of our system. We use a parameterized model that extends current models of centralized file system usage, and can reflect the large scale and the sharing patterns that may arise in future distributed systems. The key features of the model are 1) a hierarchical sharing structure, 2) a classification of files according to their sharing and access type properties 3) modeling of client processes and 4) incremental trace generation.

1. Introduction

During the past decade there has been a significant improvement in the performance of computer hardware. Three trends are apparent -

- Higher CPU speeds : Mass marketed CPU chips have increased in speed from ~ 0.01 MIPS in 1978 to ~ 10 MIPS in 1987. Bill Joy has postulated that CPU speeds will double every year after 1984 ('Joy's Law') 1.
- Increasing communication bandwidths : Fiber optic media promise communication bandwidths that are around 10,000 times that of current long haul lines. (1 Gbps versus 56 kbps) (for example, see 2)
- Cheaper, larger capacity storage devices : The price to capacity ratio of both hard disks and semiconductor memories is dropping exponentially. Optical disks offer an order of magnitude increase in storage capacity. (for example, see [. Optical Disk Calabria)

These trends in hardware will have a major impact on system design. In particular, the advent of low delay (30 milliseconds coast-to-coast), high bandwidth (up to a Gigabit/second) wide area networks will allow *very large distributed systems* (VLDS) [. Anderson 1987 .] to be developed. Such systems will span millions of hosts and users, providing global access to processing and data resources.

One of the basic services in a VLDS would be protected, synchronized access to persistent storage units (files). This *very large distributed file system* (VLDFS) would allow users located at any member site of the system to access any data resource, subject to protection constraints. Thus, users of the VLDFS could share files with any number of other users.

A VLDFS could be utilized for a number of novel purposes. For example, software updates could be implemented by updating a single shared global copy. Process migration would be easier, since open file descriptors would be globally valid . Many other uses, such as shared global databases, are also obvious.

A VLDFS extends the memory hierarchy both *vertically* and *horizontally* The vertical dimension refers to the main memory - local disk - server disk - archive levels, and the horizontal dimension refers to the geographically distributed data resources that are available to a client of the VLDFS.

The design of a VLDFS poses several interesting problems. The VLDFS should provide good file access performance ; some researchers have proposed that this can be achieved by *file caching*3 , 4 , (5) , (6) Further, the VLDFS should allow file replication (for reliability), and must synchronize access to shared

files. [1] File caching interacts with these mechanisms : for example, if a file is cached, then an update to a cached version has to be synchronized with respect to updates on other cached versions.

The problems of updating distributed cached objects are well known 7 , but the tradeoffs for large file systems have not been thoroughly studied. Further, the interactions between the caching mechanism, the replication mechanism and the synchronization mechanism are not clear. Thus, before implementing a VLDFS that provides these facilities, it is desirable that algorithms and tradeoffs in its design be analyzed.

2. Analysis Strategies

A first approximation to the general problem described above is the file allocation problem (FAP) 8 This problem is to decide which files to place at each storage node of a distributed file system in order to optimize a given performance metric. An optimal file allocation at a given moment in time depends upon the current distribution of files in the system, and the current workload.

File allocation can be static or dynamic. In the static case the allocation is computed periodically, and in the dynamic case an allocation policy is used to immediately respond to changes in the workload. File migration 9 is a subproblem of the FAP : a file is migrated to transform a previous optimal assignment to the current optimal assignment in response to a change in the workload.

There are two broad approaches to solving the FAP - analytical methods and simulation.

2.1. Analytical Approaches

where the FAP was first proposed, provided an integer programming solution . This early work was extended by 11 , 12 , 13 , 8 who used similar mathematical analysis, but with more realistic cost functions.

Queuing network models to solve the FAP have been proposed by 14 and 15

The main difficulty with analytical methods is that mathematically tractable statements of the FAP are overly simplified and so analytical solutions are not realistic. Such solutions also tend to be computationally expensive; thus they are suitable only for static allocation.

2.2. Simulation

Simulation is a better design evaluation strategy than mathematical analysis, since allocation policies can be realistically simulated. Simulation can be trace driven or synthetic workload driven. The trace driven approach has been used by 9 , 16 , 17 , 7 and 18 Results obtained using trace driven simulations are heavily dependent on the trace used, which may not be representative[2]. Further, since it is only possible to trace *existing* systems, it is difficult to use this approach in the analysis of VLDFS that have not yet been implemented.

Synthetic workload driven simulation allows one to overcome many of these limitations 19 Since the workload is under control of the experimenter, it is possible to model future workloads, and the disadvantages of trace driven simulation are avoided.

Thus, it seems clear that the most general and flexible way to analyze design problems in a VLDFS is to use synthetic workload driven simulation 20 However, in order to generate a synthetic workload, it is necessary to model the workload of a VLDFS. This is the motivation behind our work.[3]

---

1. Caching differs from replication in that caching is a *performance* optimization technique, and is workload sensitive. Replication is typically workload independent and is used to provide *reliability.*

2. For example, 16 admits that several results of his trace driven simulation may be artifacts of the trace and may not hold in general.

3. We propose to use this model along with a simulator to evaluate various VLDFS design alternatives.

3. Previous Work in Workload Modeling

This section briefly discusses previous attempts at file system workload modeling.

discusses a *driver* used to run a file system simulation. The driver implicitly defines a file system workload model. The workload model does not consider file sharing or file size.

proposes a workload model for use with a queueing network design analysis approach. However, this model is specific to a transaction processing workload.

describes a workload model for a central file system, that considers file sizes and relative frequencies of file operations. However, it ignores file sharing, and does not model file system clients.

[. Williamson Bunt .] describes a LRU stack approach to modeling file system traces. The model does not deal with the abstraction of file systems clients, file size modeling or sharing issues.

As far as we know these are the only attempts made so far in terms of file system modeling. We find these attempts ad hoc, incomplete and unsatisfactory. It is our opinion that they cannot serve as a basis of synthetic workload generation for a VLDFS.

In subsequent sections we will describe our workload model in detail. We believe that our model has overcome many of the deficiencies that we describe above.

4. Workload Model

Our workload model is unique in that we explicitly model

- file system clients
- file sharing between clients
- file reference locality, and
- file types

In this section we describe the details of our model. We also discuss how the model can be used as the basis of a synthetic trace generation algorithm.

4.1. Outline of the Model

The overall outline of our model is presented in figure 1. This figure presents the workload model as well as the parameters that synthetic workload generation requires. We partition the workload model into two submodels : the client submodel and the service submodel respectively. This partition corresponds to our view of the file system as a *service* in a VLDS, to which *clients* can make resource requests.

4.2. Service Submodel

Our model delinks the *physical* layout of the file system from its *logical* view. Whereas files may be replicated, cached and present in main memory or disk, logically a client sees a uniform open/read/write/close interface[4]. Hence, the service submodel models only the logical view of the file system.

The service submodel is described by the size distribution of the files, the age distribution of files, the set of file types, and the sharing model.

4.2.1. Size Distribution

The size distribution is a distribution describing the number of files of each size in the file system. A file is assumed to consist of an integral number of blocks. The distribution is described by a function f(x),

4. The *implementation* maps the logical file system view onto the hardware such that a given performance criterion (such as mean response time) is optimized, and given correctness criteria (such as file availability in the face of k disk failures) are satisfied.

such that x assumes integer values and if there are S files in the system, the number of files of size x blocks is Sf(x).

The choice of function f(x) is determined empirically . From our observations of files sizes in large (single computer) file system in UC-Berkeley over a period of one year, we found that a good choice of f(x) is Kx^{-m}, where K and m are fitting parameters. This basically implies that a very large fraction of files are small, and there are very few large files in the system. From the file tracing data presented in 23 , 24 , 25 , 26 , we feel that this distribution is typical of many file systems. We hence expect VLDFS files also to have a similar file size distribution.

4.2.2. Age Distribution

The age of a file is the time since it was last referenced. [5] The age distribution is modeled in the same way as the size distribution. The traces referred to earlier, as well as our own observations lead us to believe that the choice of f(x) as Kx^{-m} is a good approximation to the age distribution. Again, this is merely a way to state that most files have a short age. This behavior is consistent with the claim made by [. ousterhout trace-driven .] that most files are created and deleted within a very short space of time.

4.2.3. File Types

A file system is used in one of several ways

- as a repository for data too large to fit in the main memory
- as a long term storage for persistent data
- as the site of storage for executables
- for storing logs of system usage for accounting and monitoring

Files used for each of these distinct functions have different reference and sharing characteristics 20 , (23) Based on these distinctions, we constrain all files in the VLDFS to belong to one of four types - temporary (TMP), binary (BIN), logging files (LOG) and user data (USR). We now briefly describe the characteristics of each file type.

Temporary files are created to hold data that cannot fit all at once in the main memory, or is data that is kept briefly on disk as an insurance against main memory crashes. Examples are write ahead logs created by transaction managers, compiler intermediaries and swapped pages [6] Temporary files are characterized mainly by their short lifetimes. They usually are written to once, read once, and then deleted. They are typically not shared by clients.

Binary files store executable images. They are read from often, written to rarely, and typically shared by a number of clients. System data files, such as those that store manual pages also exhibit this behavior.

Log files are used to keep a permanent account of system usage. They are written to often, and read rarely. They are usually shared by several clients.

User files store persistent client data. They are read from and written to equally. They are typically not widely shared.

4.2.4. Sharing Levels

In a VLDFS, files will be shared by many clients. We expect the file sharing behavior to reflect the hierarchical organization of clients in the system (hence the arrow linking file sharing with the client organization).

---

5. This has also been called the f-lifetime 26

6. The Sprite virtual memory manager uses the file system to store swapped out pages. 4 explains in some detail why this is desirable.

In our model clients are clustered into groups and organizations. Examples of groups are a research group working together on the same project, and the members of the accounting department in a company. A group is characterized by the fact that the members of a group will share data amongst themselves more extensively than with members of any other group.

Similarly, we model groups as clustering together to form organizations. File sharing in organizations is constrained in the same way as file sharing in groups.

Thus, some files will be shared all the clients in the system, some files will be shared by all members in an organization , others by all the members of a group. This can be modeled naturally using a hierarchical structure. (figure 2). A file at any node in the tree can be referred to by all the clients in the subtree it subtends. Thus, a file at any node is shared by all the clients in the subtended subtree.

The fraction of files of each type at each level in the file system is a parameter to the synthetic workload generator.

4.3. The User Submodel

The previous section describes how we have modeled the file service. To completely model the workload, we have to model the users who generate the references that constitute the workload. This section describes how users are modeled.

We have described above a hierarchical structure that can model the sharing behavior of users. Thus, one element that describes users is how they are placed relative to each other in the sharing structure. This is uniquely specified by a (organization name, group name, user name) triple.

The end (human) users of the file system are modeled as a collection of processes. Thus, the actual client of the file service is a process that operates on behalf of a user who invokes it. A process generates file system requests as part of its life cycle described below.

4.3.1. Process Model

A process is described by the sequence of states it goes through in its life cycle, (the *state model*), the file operations it can perform, and the constraints it obeys when carrying out a file reference, (the *reference model*).

4.3.1.1. Operations

The file operations that a process can invoke are opening a file for reading or writing, read, write, and closing a file. We do not model file creation and deletion operations for reasons detailed in section 6.

4.3.1.2. State Model

We model a process as a closed Markov chain, where each state corresponds to an activity (figure 3). Branching probabilities are associated with each arc, these are the probabilities that the arc will be traversed during a state transition.

The state model describes a typical process birth-activity-death sequence. A process is created on behalf of a user (birth state), after which it carries out computation and I/O on behalf of the user. We are interested only in the I/O behavior of the process. Hence, we clump together all the computation into a single compute state.

We assume that I/O is carried out to files that have to be opened before use, and closed afterwards. The open and close operations delimit a *file access session*. After creation, the process enters the open_r and open_w states, in which files are opened for reading and writing respectively. Following this, a sequence of reads and writes are carried to the open file. We assume that these occur in read-write cycles, that is a number of reads cycles occur, followed by write cycles. Each read (and write) cycle in itself is a series of read-compute-read states (refer to figure 3). A process is assumed to block after each read or write request, till the request has been completed. We assume that the compute times are exponentially distributed.

Before termination, the process closes all the files (close state). It is then killed (death state). The next process is subsequently generated when the birth state is entered. There is a limbo state between the death and birth states. By associating an exponential delay with this state we model process creation as a Poisson process.

The parameters that describe the state model are the branching probabilities, the compute time distribution, and the limbo time distribution.

4.3.1.3. Reference Model

In the state model above, we describe how a process makes file service requests during its existence. The reference model describes the manner in which the requests are distributed amongst the files in the system.

We would like the reference model to be such that the service model is *stateless.* We will clarify this with an example. Let a process P1 do a 'delete' on a file at some time T. Then, no other process P2 can make a read or write reference to the file after time T, since the file no longer exists. Thus, the reference stream of P2 is influenced by the state of the service created by P1.

There are two reasons why we would like to avoid state in the service model. The principal one is that in the synthetic trace the notion of absolute time is absent. A process blocks till its read or write request is satisfied, and the amount of time it blocks depends upon the policy used to carry out the read or write (for example, a write to a replicated file may cause a number of updates, which can take a long time). Thus, at the time of trace generation, we do not know the absolute time. So, if a state change occurs at some time T, this is time relative to that process, and no other process will be able to determine the absolute time of occurrence of the state change. Thus, they cannot respond to the state change, since they are not aware of when it occurred.

To continue with our example above, process P2 will not know when the file was deleted, since it cannot relate time T to its relative time at the time of trace generation. Hence, we would like processes to generate references such that the file service model does not change state because of the references, that is, we would like the reference model to be stateless.

The other reason to choose a stateless reference model is that it allows incremental trace generation. If we would like to estimate the influence of an extra process in the workload upon a performance metric, we can simply add the reference stream due to that process to the existing workload. Incremental trace generation is hence a very useful property.

We make the assumption that a file can be referenced only if has been opened. Thus, read, write and close references must be directed to files that are already open. Therefore, the reference model is partitioned into two : a description of how 'open' operations are distributed over the set of *all* files, and a description of how the other operations are distributed amongst the set of *open* files.

4.3.1.3.1. References to Open Files

The basis for our modeling of references to open file is the file reference tracing work described in 24 and 23 These studies came to two main conclusions about file referencing

- file accesses are usually sequential
- typically the whole file is accessed (read from or written to)

Thus, in our model, we assume that all the blocks in all the open files are accessed, and these are accessed sequentially. However, we make a distinction in the case of write access to log files. Since log files are usually appended to, we assume that a write to a log file updates only one block of the log file[7].

7. The block to write to is chosen at random. This preserves the *stateless* property of the service model, which is important for incremental trace generation

As a simplification, we assume that the accesses to the open files are done in round-robin fashion. This assumption is not justifiable: the approach was chosen because of its simplicity.

4.3.1.3.2. Distribution of File Opens

In the open_r and open_w states a process opens files that are subsequently referenced by the read, write and close operations. This section discusses how we model the distribution of file open requests amongst the set of files in the system.

The basic problem that we are trying to solve is to be able to select the files that are opened so that the reference stream generated exhibits the properties of locality and sharing that are found in tracing data. We model the distribution of file opens in terms of the type selection model, the level selection model and the file selection model.

The **type selection** model assumes that each file type has a fixed fraction of opens for reading and writing associated with it. The justification for this is that the file types differ in their relative read/write frequencies. For example, BIN files are read often and rarely written. Thus, the mapping function allows us to explicitly incorporate this modeling assumption into synthetic workload generation.

The **level selection** model models the distribution of opens to files in the sharing hierarchy. It maps a reference to a file type to a sharing level. This provides a control over the amount of sharing that is present in the reference stream. For example, if we map all opens to TMP files to the unshared level, this corresponds to a workload for a file system where temporary files are not shared.

The type selection model and the level selection model allow us to map from an open reference to the level and type the reference is directed to. We now model which file at that level and of that type the reference would be made to. This **file selection** differs for BIN, LOG and USR/TMP files.

BIN files

, 20 , 27 and our own observations lead us to conclude that a small subset of BIN files are referenced very often whereas the majority of bin files are referenced rarely. To model this, we use the 60th percentile and 90th percentile of the file reference distribution to mark off BIN files. Then, with 0.6 probability, a file within the 60th percentile and with 0.3 probability, a file between the 60th and 90th percentile is chosen.

USR and TMP files

On the basis of file reference traces, 28 , 9 and 16 have claimed that file reference streams directed to user and temporary files exhibit *locality.* This means that the probability of reference to a level of the LRU stack decreases with depth 29 Hence, file selection for usr and tmp files is done on the basis of locality of reference. We assume that of the set of files that have been closed, the file closed the most recently (nearer the top of the LRU stack), has the most probability of reference. Thus, we maintain an LRU stack for each type/level pair, and select a file on the basis of LRU stack probabilities.

LOG files

There has been very little study of reference behavior to log files. However, we do not expect log files to have either locality or well defined percentile cutoffs. Thus, we select a log file at random from the set of log files at a selected level.

5. Synthetic Trace Generation Algorithm

In this section, we demonstrate how the parameters of the workload model are used to generate a file reference stream.

It is important to note that in our model there are no time dependent interprocess dependencies. This allows us to create the trace a process at a time. This makes the workload generation task incremental.

The basic approach is to simulate the process life cycle a state at a time. The branching probabilities are used to randomly decide what would be the next state of the process.

When a process enters an open state, type selection, level selection and file selection models are invoked to generate a reference to a file that is consistent with these models. This file is placed in an open_file list. When a process enters a read or write state, a file from the open_file list is selected as described above, and a reference to that file produced. This process is repeated till a trace of specified length is produced.

6. Advantages

This model offers several advantages over existing ones. We explicitly model clients, and we represent file sharing and file reference locality. Further, the reference stream does not make assumptions about the speed of file system hardware.

Client modeling using process life cycles is a big advantage. This is because we expect future systems to have life cycles similar to the one we model, the change being that compute times will be of shorter duration because of increased CPU speeds. Further, in this model we are able to control the referencing behavior, such as the number of files opened, quite accurately.

Many studies 30 , 23 have found that the amount of file sharing in current systems is quite low. This has been taken as a cue for system designers such as 31 to conclude that mechanisms that make sharing expensive are acceptable. We believe that this view is short sighted. Eventually, advances in distributed computing will inevitably lead to greater degrees of sharing. Our workload **explicitly models sharing** , and hence the sharing level in the synthetic workload generated can be controlled. This will allow us to study the behavior of file placement and management policies under different degrees of sharing.

[. Deshpande .] claims that **locality** is an important file reference property. We have incorporated notions of locality in our model.

Our model allows us to generate the synthetic trace a process at a time. That is, the workload component due to a process is independent of other processes. This allows us to generate the trace incrementally. This is very useful, since this implies that if a new process is added to the workload, or if a process is deleted, we will not need to generate the rest of the trace again.

Finally, unlike the model in .[driver .] our model does not presume to know how long it takes for a file operation to complete. Satyanarayanan has assumed that file operations arrive hyperexponentially. This implicitly assumes the response time distribution for the file system. This is not acceptable, since the aim of the performance evaluation is itself to determine the response time.

7. Drawbacks

There are three main drawbacks in our model - we have not modeled file creation and deletion, the sharing controls are not precise, and we have not validated our model.

File creation and deletion change the state of the file system permanently. Since we would like our reference model to be stateless, we have had to disallow file creation and deletion.

The other drawback is with modeling file sharing. [. Sprite virtual memory .] claims that there are two kinds of file sharing : concurrent and serial. In our model, we can control the overall degree of file sharing but cannot specify the exact proportions of concurrent and serial sharing. We believe that this is a problem. Our solution is to monitor the degree of each kind of sharing, so that we at least be able to match the degree of each kind of sharing with performance data.

Finally, since our workload model is aimed at distributed systems of the future, we have not been able to come up with traces against which our model can be validated. This is certainly a serious drawback. However, most of our modeling decisions are based upon observations that are common to a number of trace studies. Thus, we feel that the model is at least believable, if not validated.

8. Conclusions

Overall, we think that our workload model is more sophisticated than the models discussed earlier. The major features in our model are the explicit modeling of client processes, the modeling of sharing and the possibility of incremental trace generation.

We have been able to solve many of the shortcomings that we pointed out in earlier work. However, there are some drawbacks to our approach, but we believe that these are only second order effects that will not substantially alter our simulation results.

Further work is desirable in validation of the model. Our future research will be directed to this end.

References

1. M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The Design of Express," *Unpubished internal report, Dept. of EECS, UC-Berkeley*, (February 1988).
2. Pierre Halley, "," in *Fibre Optic Systems*, John Wiley and Sons, Chichester (1987).
3. M. Satyanarayanan, J.H. Howard, D.A. Nichols, R.N. Sidebotham, A.Z. Spector, and M.J. West, "The ITC Distributed File System : Principles and Design," in *Proceedings of the 10th Symposium on Operating System Principles*, (December, 1985).
4. M. Nelson, "Virtual Memory for the Sprite System," UCB Technical Report 86/301, CS Division, Dept. of EECS, Berkeley CA 94720 (June 1986).
5. L.F. Cabrera and J. Wyllie, "QuickSilver Distributed File Services : An Architecture for Horizontal Growth," RJ 5578 (56697), CSD, IBM Almaden Research Center, San Jose, CA 95120 (April 1987).
6. A.J. Smith, "Problems, Directions and Issues in Memory Hierarchies," *Proc. 18th Annual Hawaii International Conference on Systems Sciences*, pp. 468-476 (Jan 1985).
7. C. Kent, "Cache Coherence in Distributed Systems," in *PhD Thesis*, Purdue University (August 1986).
8. L.W. Dowdy and D.V. Foster, "Comparative Models of the File Assignment Problem," *Computing Surveys*, (June 1982).
9. A.J. Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms," *IEEE TOSE* **SE-7**(4)(July 1981).
10. R.G. Casey, "Allocation of Copies of a File in an Information Network," *AFIPS SJCC*, pp. 617-625 (1972).
11. H.L Morgan and K.D. Levin, "Optimal Program and Data Locations in Computer Networks," *CACM*, (May 1977).
12. E. Grapa and G.G. Belford, "Some Theorems to Aid in Solving the File Allocation Problem," *CACM* **20**(11)(November 1977).
13. C.V. Ramamoorthy and B. Wah, "Data Management in Distributed Databases," *Proc. of NCC 1979*, p. 667 (1979).
14. D. Ferrari and T.P. Lee, "Modeling File System Organizations in a LAN Environment," UCB Technical Report 83/142, CS Division, Dept. of EECS, UC Berkeley, Berkeley, CA 94720 (1983).
15. K.K. Ramakrishnan and J.S. Emer, "A Model of File Server Performance for a Heterogenous Distributed File System," *Proc. of the 1986 SIGCOMM*, pp. 338-347 (August 1986).
16. M.B. Deshpande, "Locality-Based Approached to Dynamic File System Management," 85-14, Dept. of Computational Science, University of Saskatchewan, Saskatoon, Canada (1985).
17. J.B. Porcar, "File Migration in Distributed Computer Systems," in *PhD Thesis*, University of California, Berkeley (December, 1982).

18. D. H. Lawrie, J. M. Randal, and R. R. Barton, “Experiments with Automatic File Migration,” *IEEE Computer*, pp. 45--55 (1982).

19. D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems*, Prentice Hall, Englewood Cliffs, NJ (1983).

20. M. Satyanarayanan, “A Synthetic Driver for File System Simulations,” Unpublished Report (Part of PhD Thesis), ITC, CMU, Pittsburgh PA 15213 (1983).

21. M. Satyanarayanan, “,” in *Modelling Storage Systems*, UMI Research Press (1986).

22. A. Park and R.J. Lipton, “Models and Measurements of File System Performance,” CS-TR-067-86, Princeton University (December 1986).

23. R. Floyd, “Short Term File Reference Patterns in a UNIX Environment,” TR 177, University of Rochester (March 1986).

24. J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, “A Trace-Driven Analysis of the Unix 4.2 BSD File System,” pp. 15--24 in *Proceedings of the 10th Symposium on Operating System Principles*, ACM (December, 1985).

25. D. Gawlick, “Measurements of File System Workload at Amdahl,” *Presentation at UC-Berkeley*, (October, 1987).

26. M. Satyanarayanan, “A Study of File Sizes and Functional Lifetimes,” *Proceedings of the 8th Symposium on Operating System Principles*, pp. 96--108 (December, 1981).

27. B. Pope, “Dynamic Access of CMS Files,” RC 10483 (#46844), IBM T.J. Watson Research Center, NY 10598 (April 1984).

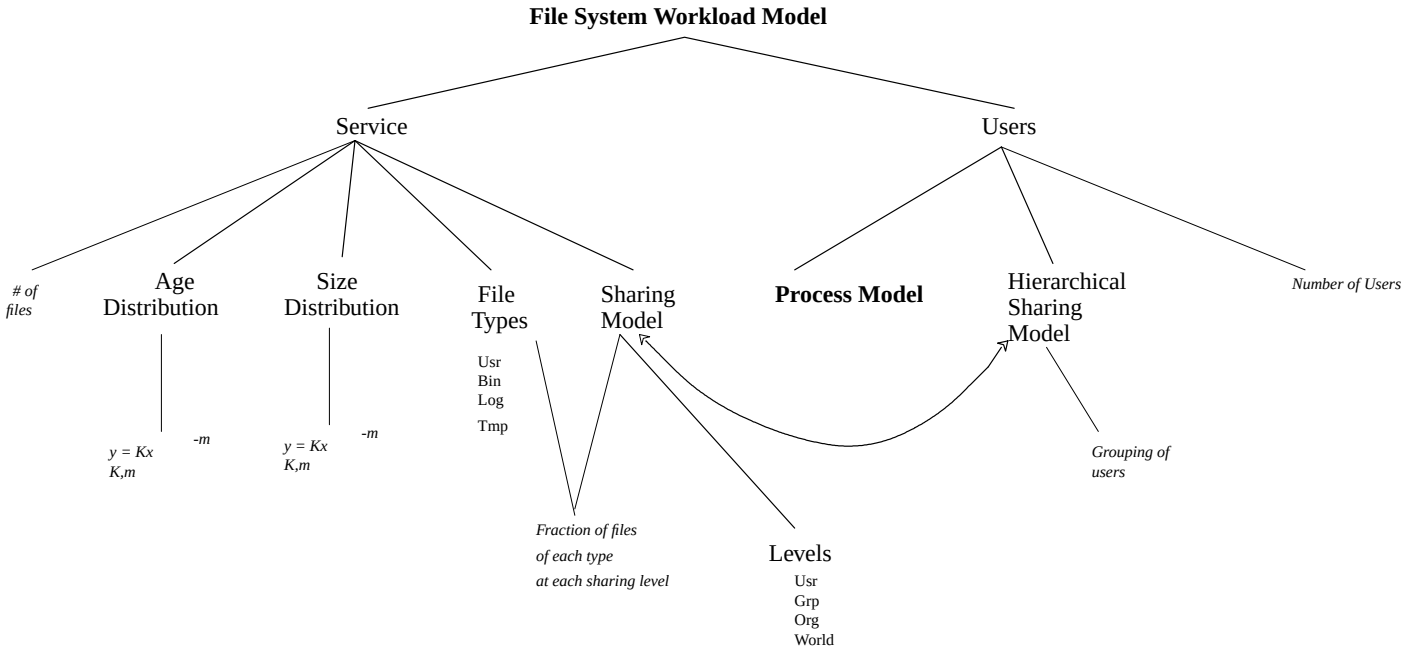
28. C.L. Williamson and R.B. Bunt, “Characterizing Short Term File Referencing Behavior,” 85-6, Dept. of Computational Science, University of Saskatchewan, Saskatoon, Canada (July 1985).

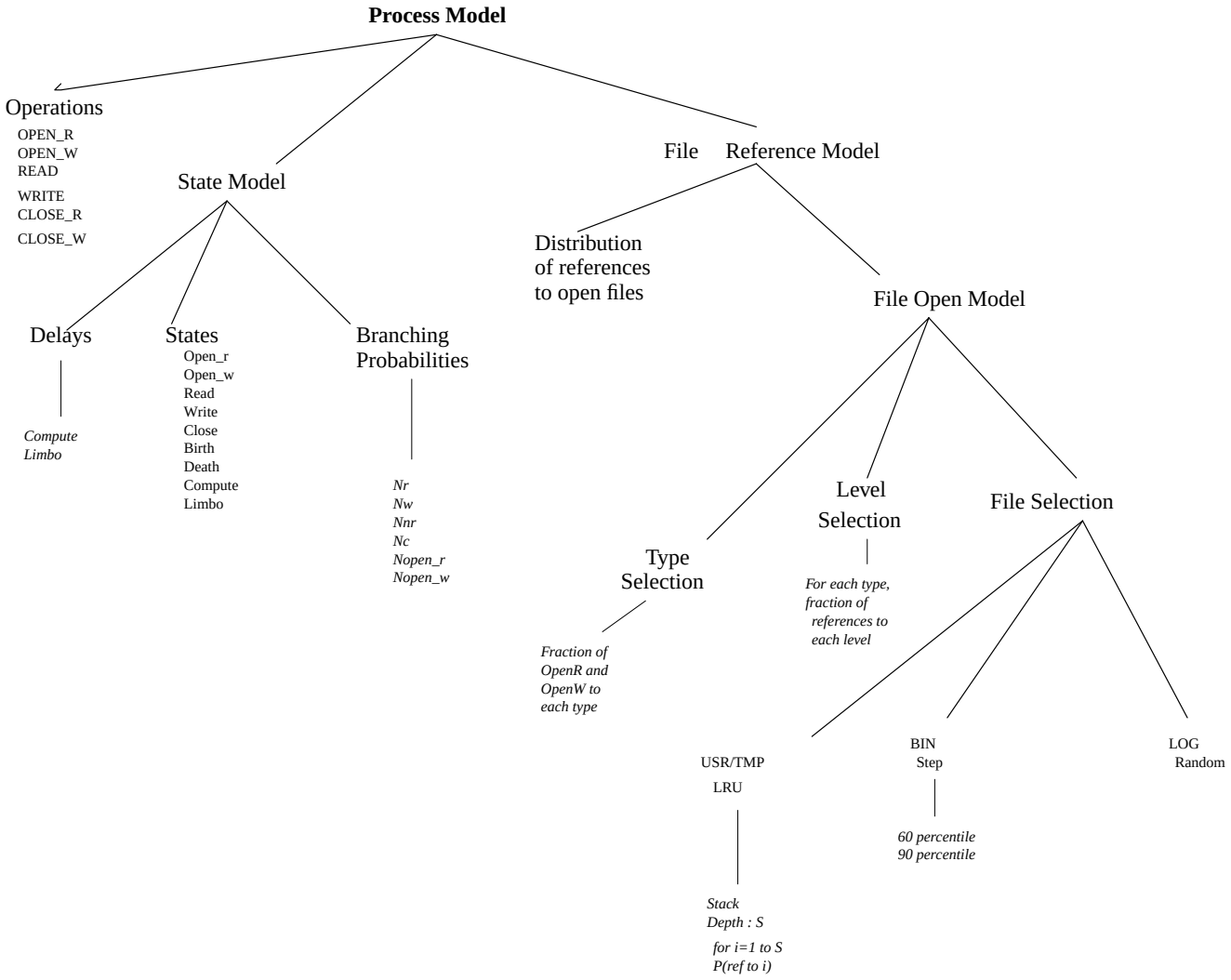
29. P.J. Denning, “Working Sets Past and Present,” *IEEE TOSE* **SE-6**(1) pp. 64-84 (January 1980).

30. W.A. Montgomery, “Measurements of Sharing in Multics,” *Proceedings of the 6th SOSP, OS Review* **11**(5) pp. 85-90 (November, 1977).

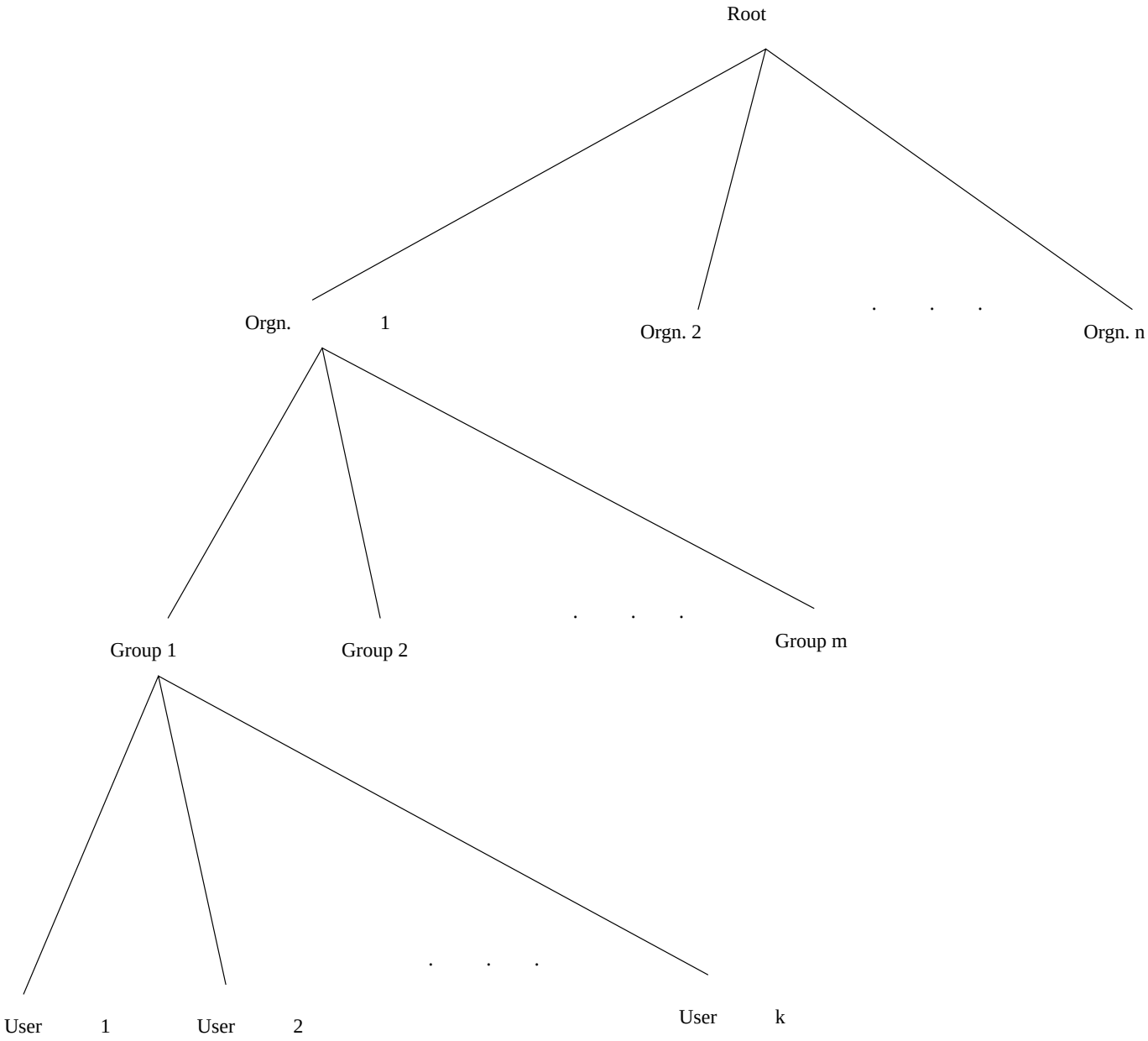
31. M. Nelson, B. Welch, and J. Ousterhout, “Caching in the Sprite Network File System,” *ACM TOCS*, (to appear). Originally presented at the *Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, 8-11 November 1987

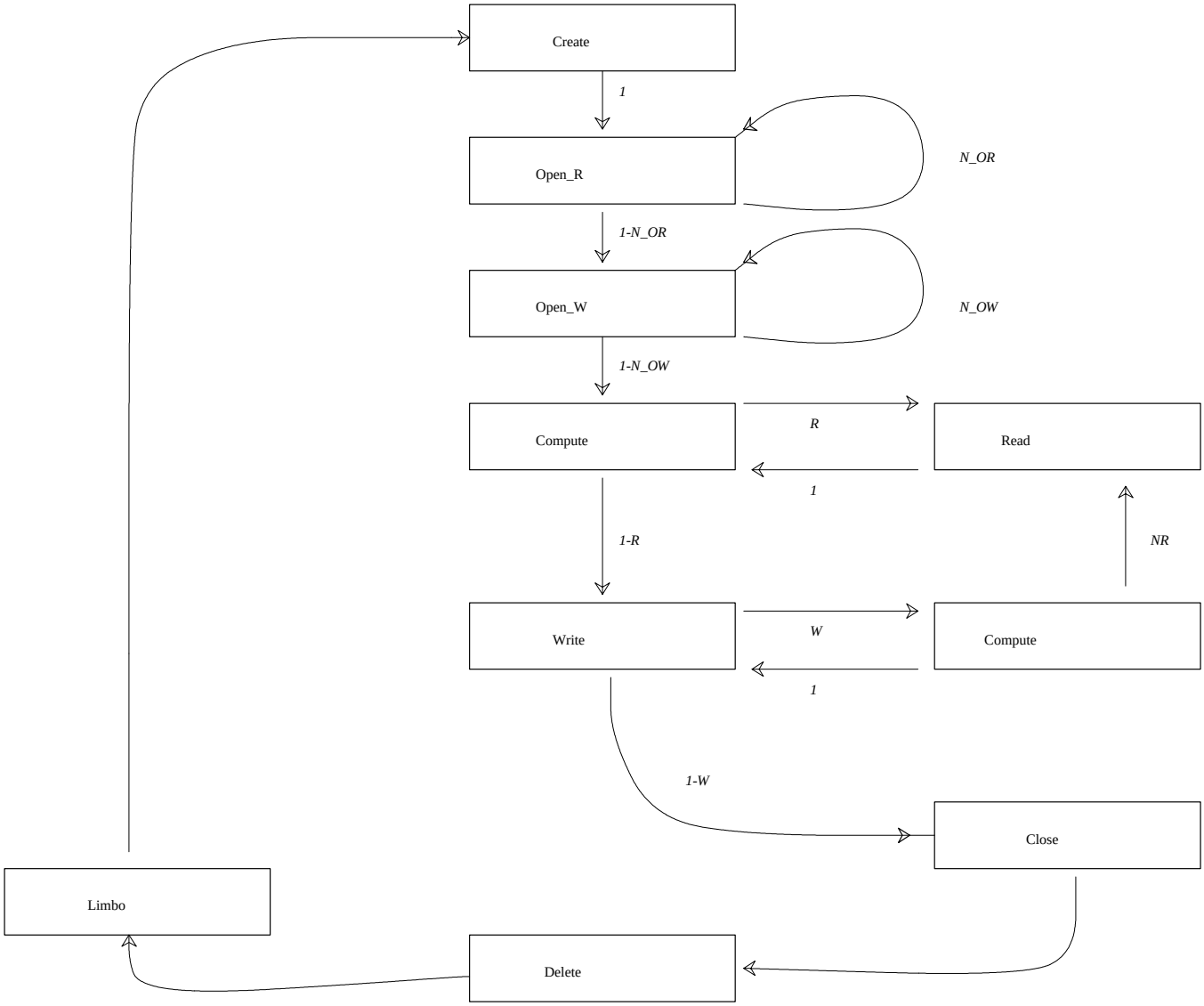
Figure 1


File System Workload Model
Service
Users
# of files
Age Distribution
Size Distribution
File Types
Sharing Model
Process Model
Hierarchical Sharing Model
Number of Users
y = Kx -m
K,m
y = Kx -m
K,m
Usr
Bin
Log
Tmp
Grouping of users
Fraction of files of each type at each sharing level
Levels
Usr
Grp
Org
World



Parameters to the synthetic workload generator are italicized


Root
Orgn. 1
Orgn. 2
. . .
Orgn. n
Group 1
Group 2
. . .
Group m
User 1
User 2
. . .
User k



Tags on arcs represent branching probabilities