UIDS as Internal Names in a Distributed File System

Paul J. Leach, Bernard L. Stumpf,
James A. Hamilton, and Paul H. Levine
Apollo Computer, Inc.
19 Alpha Road, Chelmsford, MA 01824

Abstract

The use of UIDs as internal names in an operating system for a local network is discussed. The use of internal names in other distributed systems is briefly surveyed. For this system, UIDs were chosen because of their intrinsic location independence and because they seemed to lend themselves to a clean structure for the operating system nucleus. The problems created by UIDs were: generating UIDs; locating objects; supporting multiple versions of objects; replicating objects; and losing objects. Some solutions to these problems are presented; for others, no satisfactory solution has yet been implemented.

1. Introduction

Although the area of distributed systems is a relatively new one, there are already many examples of implemented distributed operating systems for local networks and their attendant file systems. Many of these systems have chosen to use internal names for the objects they support, into which user visible text string names are mapped. Among the most popular forms of internal name have been unique identifiers (UIDs); however, there has been little in the literature discussing the motivation for choosing one form of name over another, or the consequences of a choice once made. This paper presents the experiences that resulted from using UIDs as internal names in one particular distributed system: the Aegis operating system for the Apollo DOMAIN network [APOL 81, NELS 81].

1.1. Organization

The rest of this paper is organized as follows. Section 2 discusses internal names as they are used in several other distributed systems. Section 3 presents an overview of the DOMAIN system environment, and of the nature of UIDs and objects in Aegis. Section 4 deals with the motivations and perceived advantages that led us to choose UIDs. Section 5 deals with the problems we foresaw or discovered in the process of implementing the system, and presents some solutions to these problems. Section 6 offers some final observations and conclusions.

2. Internal names in other systems

Given that one decides to use internal names, there seem to be just two fundamental alternatives: to use UIDs or "structured names". UIDs can be thought of as simply large integers or long bit strings, although some other information may be encoded within them. The important characteristic is that they are large enough that the same UID will never refer to two different objects at the same time. Structured names, as in [SVOB 79], contain more than one component, some of which are used to indicate the location of, or route to, the object named. However, individual components may be unique for all time only within the context of the other components; some systems with this property have called their internal names UIDs. This section briefly indicates the internal naming schemes used by several distributed systems or their distributed file system components.

2.1. WFS

The Woodstock File Server (WFS) [SWIN 79] uses "file identifiers" (FIDs) to name files. FIDs are 32 bit unsigned integers, which are unique for all time within a individual WFS server, but may be duplicated across servers. Thus, it is up to each WFS client to remember the server associated with each FID. The combination of server name and FID is a form of structured name. The mapping from FID to physical disk addresses is via a hash table.

2.2. Pilot

Pilot [REDE 80] uses "universal identifiers (UIDs)" to name files; they are 64 bits long and "guaranteed unique in both space and time". UIDs were chosen so that removable volumes could be transported between machines without fear of conflict. A B-tree is used to map UIDs to physical disk addresses.

2.3. DFS

The distributed file system (DFS) [STUR 80] also uses UIDs. We suspect that they are really UIDs because the implementors provide "a simple locating service" to help find the server which holds a file, given only its UID; a structured name would not need a locating service. Like Pilot, a B-tree is used to map UIDs to physical disk addresses.

2.4. CFS

The Cambridge File Server (CFS) [DION 80] uses what it calls UIDs to name files. They are 64 bits long; 32 bits are a random number, and 32 bits contain the disk address of the object's descriptor. The use of garbage collection [GARN 80] guarantees that an object will not be deleted while a reference to it exists, and therefore that, within a single server, a UID can never refer to more than one object. However, it seems that UIDs can be duplicated on different servers, although the 32 bit random number makes it highly improbable.

2.5. Felix

The Felix File Server [FRID 81] uses a system generated "File Identifier" (FID) to name files. AN FID is a "universal access capability" for the file it names. When the file is deleted, its FID is guaranteed not to be reused for a certain period of time. It also seems that FIDs with the same numerical value can be in use by more than one server at the same time.

2.6. LOCUS

The LOCUS system [POPE 81] uses structured internal names. A name is a pair "<file group number, file descriptor number>". The file group number can be thought of as uniquely identifying a logical volume. The file descriptor number is an index into a per-file-group array of file descriptors; it is unique within a file group as long as any references to the file it identifies exist. The choice of internal name seems to have been motivated by UNIX (TM, Bell Laboratories) compatibility constraints: directory structures are visible to application programs and contain file descriptor numbers, which are relative to the file group containing the directory.

2.7. Others

There are a number of other recent implementations of, or designs for, distributed systems for which descriptions have been published: S/F-UNIX [LUDE 81]; ACCENT [RASH 81]; TRIX [WARD 80, CLAR 81]; EDEN [LAZO 81]. However, they concentrate on other aspects of distributed systems design, and do not provide much information on their use of internal names.

2.8. Summary

When the design of Aegis began in early 1980, there were fewer examples of distributed systems to study; Pilot and WFS particularly influenced us. Pilot uses UIDs; WFS uses IDs which are unique within a single file server, but which require its clients to remember upon which server files reside. From our studies we got little motivation for either choice; yet upon starting our design it became clear that there were non-trivial problems involved with either choice.

3. DOMAIN system environment

3.1. Hardware

A DOMAIN system consists of a collection of powerful personal computers (nodes) connected together by a high speed (12 megabit/second) local network. Each node has a 'tick' time [LAMP 80] of 1.25 microseconds and can have up to 3.5 megabytes of main memory. Most nodes have 33 megabytes of disk storage and a 1 megabyte floppy disk, but no disk storage is required for a node to operate. A bit mapped display has 800 by 1024 pixels, and a bit BLT (block transfer) to move arbitrary rectangular areas at high speed. The display is allocated into windows (called PADs) which are a form of virtual terminal [LANT 79]; multiple concurrent processes, each possessing its own window(s), can be controlled by the user simultaneously. Dynamic address translation hardware allows each process to address 16 megabytes of demand paged virtual memory. The network arbitrates access using a token passing method; each node's network controller provides a unique node ID which is assigned at the factory and contained in the controller's microcode PROMs.

3.2. System usage characteristics

It is expected that the nodes in a network will be owned by many organizations, with each organization owning many nodes. One organization is likely to be chartered to provide computing related services and resources to the entire network community. Within an organization, a high degree of cooperation will be desired; while between organizations, a higher degree of autonomy will be preferred; and the service organization wants resource sharing, protection and (perhaps) accountability. Aegis provides tools to allow a high degree of cooperation, and tools to create policies which can allow a high degree of autonomy. This results in an environment of "policy parameterized autonomy".

3.3. Objects and UIDs

At the highest level, Aegis is an "object-oriented" system, and objects are named by UIDs. Objects are typed and protected:

associated with each object is the UID of an access control list, the UID of a type descriptor, as well as a physical storage descriptor, and some other attributes. Supported objects include: alphanumeric text, record structured data, IPC mailboxes, executable modules, directories, access control lists, serial I/O ports, magnetic tape drives, and display bit maps. UIDs are also used to identify persons, projects, and subsystems for protection purposes.

Aegis UIDs are 64 bit structures, containing a 36 bit creation time, a 20 bit node ID, and 8 other bits whose use is described later. UIDs possess the addressing aspects of a capability, but without the protections aspects [FABR 74]. Or, a UID can be thought of as the absolute address of an object in a 64 bit address space. The hardware does not support this form of address, so programs access objects by presenting a UID and asking for the object it names to be "mapped" into the program's hardware processor address space (see [REDE 80] on the desirability of mapping in distributed systems). After that, they are accessed via virtual memory paging: not to create shared memory semantics, but as a form of lazy evaluation, since only the needed portions of objects are actually fetched from disk or over the network.

The system provides a high degree of network transparency in accessing objects. The mapping operation is independent of whether the UID is for a remote or local object. As long as programs assume that their objects are not local, and hence operations on them are subject to communication failures, they need not be aware of their location (see [POPE 81] for a discussion).

3.4 Naming objects

Text string names for objects are provided by a directory subsystem layered on top of the Aegis nucleus. The name space is a hierarchical tree, like Multics [ORGA 72] or UNIX [RITC 74], with directories at the nodes and other objects at the leaves. Each directory is primarily a simple set of associations between component names (strings) and UIDs. The absolute path name of an object is an ordered list of component names. All but (possibly) the last are names of directories, which, when resolved starting from a network-wide distinguished "root" directory, lead to the UID of the object. Thus, an absolute path name, like a UID, is valid throughout the entire network, and denotes just one object.

4. Motivation for using UIDs

There were several main reasons for choosing UIDs as internal names. First, we wanted location independence: to divorce the internal name of an object from its location in the network. Second, we wanted absolute internal names: ones that could be passed from process to process, and from node to node, without having to be relocated at each step. Third, we wanted to separate text string naming from internal naming, in order to remove string name management from the nucleus. Fourth, we wanted a uniform way of naming all objects in the system. Fifth, we wanted to be able to construct composite objects (objects which refer to other objects) easily, and to allow user programs to do likewise. Sixth, we wanted to allow for typing of objects, and in a potentially extensible and manageable way.

We wanted objects to be able to move without having to find and alter all references to them. The system does not move objects except when explicitly directed to do so. However, users may want to move dismountable volumes from one node to another, or to move a peripheral from a disabled node to a functioning one. Structured names imply locations, which makes moving an object harder, because references to the moved object have to be updated; this in turn mitigates against composite objects. UIDs, because of their location independence, have no such problem.

From an implementation point of view, we wanted to be able to start with simple object locating algorithms, perhaps with restrictions placed on object locations, and work up to better ones, again without changing any stored data. Structured names seemed to freeze this decision too early: the locating scheme is bound into the name. We also wanted to avoid the proliferation of ad hoc internal names by having a single, simple, cheap, uniformly applicable naming scheme available at all but the lowest levels of the system.

Text string names can also be made location independent, but we wanted the nucleus interface to be simpler than string names. Also, string names are too long to be embedded in objects, too expensive to resolve, and therefore can usually be used only at fairly high levels in the system.

So, unlike structured names, UIDs had the right properties to satisfy these requirements. They are intrinsically location independent: they uniquely identify an object no matter where it resides. The node ID contained in our UIDs says where the object was created, but has no necessary connection with its current location. They are absolute, and they are (relatively) short and of fixed length. The combination of these attributes means that it is easy to embed UIDs in objects to make composite objects, and that there is little space penalty in using them to name all objects. It also makes it easy to do mapping from text string names to UIDs in a layer above the nucleus. A UID can be used to denote the type of an object. New types (UIDs) can easily be generated without interfering with others doing the same, and can extensibly refer to a type descriptor object containing type data and operations.

There were other, less crucial, advantages that we foresaw. UIDs are good for objects without string names, such as temporary files; objects can even be created as temporaries, then given string names later. Because they are short, they can be easily hashed, and stored in system tables, and passed in IPC messages.

Because they are guaranteed to be unique, they can be used as transaction IDs, with the TID also serving to name the commit record for the transaction. Finally, because UIDs are hard to guess, there are certain capability protection aspects to them: in some cases, it may be acceptable to use possession of a UID as permission to operate on the underlying object.

5. Problems with UIDs

We also quickly discovered that there were problems that needed solution to use UIDs effectively.

1. Generating UIDs and guaranteeing their uniqueness.
2. Locating an object given its UID.
3. Naming different versions of an object
4. Replication of objects
5. Lost objects

5.1. Generating UIDs

We thought that generating UIDs would be easy: concatenate the node ID of the generating node with a reading from its real time clock. The first issue to deal with was choosing the size of the UID. We had a 48 bit 4 microsecond basic system clock, but that, plus a 20 bit node ID, and a few bits for future expansion, seemed to imply a UID that we felt would be a bit long. We settled on a 36 bit creation time, which meant a 16 millisecond resolution. We justified it by noting that, since most objects reside on disk, they can't be created faster than disk speeds, 36 bits allowed a resolution several times higher. To allow for possibly bursty UID generation, the system remembers unused UIDs from the previous minute or so, and uses them before generating new ones.

The second issue is guaranteeing uniqueness. Concatenating a node ID and a real time clock reading guarantees uniqueness as long as one makes sure that the clock always advances. We thought this could be assured by providing a battery operated calendar clock from which to initialize the real time clock. But batteries have a limited shelf life; and since it is important that a UID not be reused, other measures were needed. So the system stores the last shutdown time on the disk, and checks it against the calendar clock during initialization. If the time is too far wrong, either backward, or forward, it requests verification and/or correction from the user. It is clear that the clock cannot be allowed to go backwards; what may not be so instantaneously obvious is that too long a foward jump is also dangerous. Such a jump is likely to be an error, requiring later correction; but if any UIDs are generated from the erroneously advanced clock, they may be duplicated when real time catches up to that point.

Another solution is to use other nodes in the network to corroborate the calendar clock reading; but since it is possible that none will be available, our solution would still need to be resorted to in that case. It seems that no solution is foolproof, but that the probability of failure can be made fairly small. Our experience to date supports this conclusion: with several hundred nodes in use, we know of no problems.

5.2. Locating objects

A direct consequence of the location independence of UIDs is that a locating service is needed to find an object given its UID. This is the fundamental distributed algorithm in Aegis: no global state information is kept about object locations. The complexity of this task depends on the restrictions on object location that higher levels of the system can enforce, and on the desired level of performance. Some examples of the effect of various restrictions that could be imposed are as follows.

- One can restrict objects not to move from the node where they are created, in which case node ID part of the UID is certain to be the location of the object.
- One can restrict (most) objects to be on same volume as the directory in which they are cataloged. Then, as long as the locations of a few volume root directories can be found, all other objects can be found.
- One can restrict object location as in either of the above examples, then relax it by establishing equivalence classes among nodes or volumes, such that if the above rules allowed an object to be on one node or volume of a class, then by these rules, it could be on any node or volume in the class. This would allow multiple physical copies of an object with the same UID to exist and be located.
- Of course, it is possible to have no restrictions at all, and still locate objects. After whatever other means exist have failed, a request to return the location of an object can be broadcast, and an answer awaited. Also, in this case, there is absolutely no necessary relation between nodes or volumes and directory hierarchies, making hierarchy backup and crash reconstruction difficult.

We considered all the schemes indicated by the above examples. Because we allow removable volumes, the assumption that objects reside at the node where they were created is not valid. We also convinced ourselves that in a sufficiently large (inter)network, and given the possibility of removable volumes whose node of origin was in a disjoint network, we could not guarantee to find an object even if it were online and accessible. As noted above, even in this case the object could be found if one were willing to make a broadcast to the entire internet, and wait a (possibly) very long time for an answer; but since this had performance implications, as well as the other problems noted above, we were unwilling to base our design on this approach. Thus, we would have to rely on heuristics, and, ultimately, perhaps even help from the user. Our initial goal was to pursue the second approach, as it met our immediate requirements; and it can readily be extended into the third scheme, which we think is sufficiently flexible to eliminate any need for the fourth.

We have already gone through three generations of locating algorithms, and can foresee more. They used two sources of 'hints': the node ID in the UID, and the hint manager. The sources for the hint manager's hints can be any program which believes it can guess the whereabouts of an object, or even direct input from a user. In particular, the string name manager guesses that a cataloged object is on the same node as the directory in which it is cataloged (except for special node boundary crossing points).

The first generation algorithm was very simple. To locate an object given a UID, it would first search all local disks. If the local search failed, it would try the node whose ID was contained in the UID. This procedure could always find local objects, objects on dismountable volumes mounted locally, and remote objects that had never moved from where they were created; others, however, could not be located. In particular, remote objects on removable volumes that had been moved from their creation node were unlocatable. Also, for remote objects, time was wasted searching local secondary storage. Note that for remote objects in this scheme, the node ID in the UID was more than just a hint: it had to be right.

The second algorithm added the hint manager. After trying locally, it would consult the hint manager, and if a hint were present, would use the hint. If this failed, it would proceed as in the first case. Therefore, even remote objects on removable volumes could be located, if they were on the same node as the directory in which they were cataloged. This would normally be very likely even if we didn't enforce it (which we currently do).

The time wasted searching locally for remote objects in the previous algorithms was noticeable, so a third was adopted. Before searching locally, the node ID in the UID is examined; if it is not the ID of the local node, then the local search is bypassed. Only if the remote search fails is a local search initiated.

In the future, it is likely that direct input to the hint manager will be added, as will the equivalence class technique. Also, in an internet environment, a second level of hint manager, usually residing on gateway nodes, will probably become necessary. However, its task will be eased considerably because it will only have to store location information for objects that could not be located using the other available hints.

It is significant to note that the object locating service is layered above the nucleus. An object's location is determined when it is mapped into a process' address space, and retained. Thus, it is guranteed to be known at critical junctures, such as when servicing page faults. It is also cached, so that the location of active objects is likely to be in the cache. The first case is important for clean system structure; the second for good system performance. However, even in the absence of cached or retained information, locating a remote object usually takes only one, and at most two, messages with the current algorithm.

Using UIDs, plus repeated improvement to locating algorithms, has allowed us to benefit from the location independence of UIDs, without paying a serious performance penalty.

5.3. Object versions

If UIDs are allowed to be embedded in objects, the object version problem arises. The object containing the reference may wish not to refer to a particular instance of an object, but to its latest version. A procedure object may contain the UIDs of other programs or of libraries, for example. The fundamental problem is that the same UID can not name two different objects, even if they are just different versions. (For Aegis UIDs, this is true; if they contained an explicit version number, it need not be true.) We see two possible solutions to this problem in our context, both of which involve the use of indirection objects; in one case, the indirection object contains a symbolic name; in the other, the UID of the current version of the object. (Indirection objects with symbolic names are also used in the iMAX-432 filing system [POLL 81], where they are called linkage objects.) In the first case, whenever a new version becomes available, the binding of the symbolic name is changed to refer to the new version. In the second case, the indirection object is updated with the new version's UID. In our environment, the second solution is simplest, because it doesn't involve the string name manager to resolve the reference. (The iMAX-432 uses the symbolic solution because it doesn't have real UIDs.)

5.4. Replication

To take advantage of the potential for enhanced reliability that distributed systems offer, it is desirable to be able to redundantly store objects at more than one node. The logcal object thus created we call a replicated object and each of the redundant copies we call a replica. If a replicated object is immutable, this presents no great problem. It is relatively easy for the nucleus to support a replicated immutable object: all the replicas can have the same UID. Even though this results in multiple physical objects with the same UID, since they are all immutable and identical, it never matters which one the nucleus finds and uses; there is only one logical object with that UID. One of the object attributes supported by Aegis' nucleus is immutability.

For mutable objects, however, it is not as easy; updates to the object instances must be coordinated so that all clients see a consistent state. We don't deal with the concurrency management problem here, only the problem of naming the replicated object and its components. ([GIFF 79] and [POPE 81] deal directly with replication; DFS [STUR 80] provides general support for multi-node atomic operations which can be used for replication purposes.) Because it is complex, it is desirable to leave the

management of replication out of the nucleus, while still allowing it to be conveniently layered on top. In order to make the new layer transparent to client programs, it is necessary that they be able to refer to a replicated object via one UID. The replication manager, on the other hand, needs to distinguish between the replicas, because internally to it they will have different states, even though the client only sees consistent states. Thus it needs different UIDs for each replica. This leads to essentially the same difficulty as in the object version problem: the same UID needs to refer to more than one object. The replication manager must map a UID presented by a client into the UIDs of the mutable replicas.

One way to accomplish this is to record the UIDs of the replicas in an immutable object, and have clients use its UID to denote the replicated object. A copy of this immutable object is then put at each site holding a replica. When a client refers to the replicated object, its UID is used to locate one of the immutable object copies; if one can be found, then at least the replica at the same site will be available. However, this does not allow the addition of new replicas. To solve this, we use 4 of the 8 'other' bits in the UID to denote particular replicas; let us call it the replica field. A replicated object has a UID with a replica field of zero; there is no physical object with this UID. Each of the replicas (up to fifteen of them) has the same UID except for a non-zero replica field. Thus, a client of a replicated object always names it with a UID having a replica field of zero; the replication manager selects and operates on specific replicas via non-zero replica fields.

Contrasting the two solutions, we see that using an immutable object supports an arbitrary mapping from UID of a replicated object to the UIDs of the replicas which constitute its representation; whereas the second scheme causes these UIDs to be easily computable from one another, eliminating the need for the arbitrary map. In addition, the second solution allows replicas to be added and deleted.

5.5. Lost objects

A lost object is one which exists, but for which no references exist; hence it is inaccessible, i.e. lost. Unfortunately, it still takes up disk space. Objects become lost due to crashes, or when objects which contain references to them are deleted. Actually, objects are never completely lost: a scan of a volume's (undamaged) table of contents data structure can find all objects on a volume. However, if an object becomes inaccessible via its text string name, it is often as good as completely lost. The only complete way to recover is garbage collection, but we chose not to implement it. Again, the consideration was nucleus complexity: if internode object references are allowed, a distributed, asynchronous collector is called for, such as [BISH 77]. We knew of no implemented example; the nearest thing is the CFS garbage collector [GARN 80], which is asynchronous, but which doesn't handle internode references. Furthermore, in our current objects, there is no general way to locate all the UIDs, although the implementation of partitioned objects (objects segregated into UID parts and data parts [JONE 80]) would solve this problem. Finally, we felt that most common cases could be handled without it. Most objects are cataloged; and by arranging that an object is not marked permanent until it has successfully been cataloged, any newly created but not yet cataloged object will still be temporary if the system crashes, and will be deleted by the file system salvager (see [REDE 80]). Furthermore, all objects have a father object attribute, which is the UID of the directory in which they are cataloged, or of the (primary) object which contains its UID. If the father object should cease to exist, the resulting lost object(s) can be deleted. Thus, object tree structures can be handled. We felt that the sum of these techniques would be sufficient.

6. Observations and conclusions

The principal advantages of UIDs are their size, location independence, and the opportunity for layering the nucleus implementation that they provided. Most of the problems involved have been overcome or are understood satisfactorily; the possible exception is the general lost object problem. A feature of UIDs we have taken advantage of is that, because they are location independent, initial implementations of higher layers can impose restrictions on object location, and the restrictions can later be removed without restructuring the lower layers; the same would seem to be hard to accomplish with structured names.

Of course, it is eventually necessary to translate UIDs into structured names, because the knowing the location of an object is a prerequisite to accessing it. We have found it advantageous to delay this binding as long as possible, and to make general and uniform use of the unbound names.

Aegis as currently implemented is missing some of the features described above. Presently, it does not support object replication, partitioned objects, garbage collection, network verified time for UID generation, or extensible types. However, the fundamental groundwork, that of making a design that can be gracefully extended, and anticipating the most likely areas of extension, is essential to any system which is intended to have a long and useful life. We think that we have accomplished that goal.

References

APOL 81 ----
Apollo DOMAIN Architecture. Apollo Computer Inc., Chelmsford, Mass., 1981.

BIRR 80 Birrel, A. D., Needham, R. M.
"A Universal File Server."
IEEE Tranactions on Software Engineering, SE-6, 5 (September 1980), pp. 450-453

BIRR 81 Birrel, A. D., Levin, R., Needham, R. M., Schroeder, M. D.
"Grapevine: An Exercise in Distributed Computing."
Preprints for the Eighth Symposium on Operating Systems Principles, December 1981, pp. 54-69.

BISH 77 Bishop, P. B.
Computer Systems with a Very Large Address Space and Garbage Collection. Technical Report LCS/TR-178, Laboratory for Computer Science, M.I.T., Cambridge, Mass., May 1977.

CLAR 81 Clark, D., Halstead, B., Keohan, S., Sieber, J., Test, J., Ward, S.
"The TRIX 1.0 Operating System."
Newsletter of IEEE Tech. Comm. on Distributed Processing, 1, 2 (December 1981), pp. 3-5.

DION 80 Dion, J.
"The Cambridge File Server."
Operating Systems Review, 14, 4 (October 1980), pp. 26-35.

FABR 74 Fabry, R.S.,
"Capability-Based Addressing"
Communications of the ACM, 17 7 (July 1974), pp. 403-412.

FRID 81 Fridrich, M., Older, W.
"The FELIX File Server."
Proceedings of the Eighth Symposium on Operating Systems Principles, December 1981, pp. 37-44.

GARN 80 Garnett, N. H., Needham, R. M.
"An Asyncronous Garbage Collector for the Cambridge File Server."
Operating Systems Review, 14, 4 (October 1980), pp. 36-40.

JONE 80 Jones, A.K.
"Capability Archictecture Revisited."
Operating Systems Review, 14, 3 (July 1980), pp. 33-35.

LAMP 80 Lampson, B. W., and Redell, D. D.
"Experience with Processes and Monitors in Mesa."
Communications of the ACM, 23, 2 (February 1980), pp. 105-113.

LANT 79 Lantz, K. A., Rashid, R. F.
"Virtual Terminal Management in a Multiple Process Environment."
Proceedings of the Seventh Symposium on Operating Systems Principles, December 1979, pp. 86-97.

LAZO 81 Lozowska, E., Levy, H., Almes, G., Fischer, M., Fowler, R., Vestal, S.
"The Architecture of the Eden System."
Proceedings of the Eighth Symposium on Operating Systems Principles, December 1981, pp. 148-159.

LEVI 79 Levin, R., Cohen, E., Corwin, W., Pollack, F., Wulf, W.
"Policy/Mechanism Seperation in Hydra."
Proceedings of the Fifth Symposium on Operating Systems Principles, December 1979, pp. 132-140.

LISK 79 Liskov, B.
"Primitives for Distributed Computing".
Proceedings of the Seventh Symposium on Operating Systems Principles, December 1979, pp. 33-42.

LUDE 81 Luderer, G. W. R., Che, H., Haggerty, J. P., Kirslis, P. A., Marshall, W. T.
"A Distributed Unix System Based on a Virtual Circuit Switch".
Proceedings of the Eighth Symposium on Operating Systems Principles, December 1981, pp. 160-168.

NEED 78 Needham, R. M., Schroeder, M. D.
"Using Encryption for Authentication in Large Networks of Computers."
Communications of the ACM, 21 12 (December 1978), pp. 993-999.

NELS 81 Nelson, D. L.
"Role of Local Network in the Apollo Computer System."
Newsletter of IEEE Tech. Comm. on Distributed Processing, 1, 2 (December 1981), pp. 10-13.

ORGA 72 Organick, E. I.
The Multics System: An Examination of Its Structure M.I.T. Press, 1972.

POLL 81 Pollack, F., Kahn, K., Wilkinson, R.
"The iMAX-432 Object Filing System."
Proceedings of the Eighth Symposium on Operating Systems Principles, December 1981, pp. 137-147.

POPE 81 Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., Thiel, G.
"LOCUS: A Network Transparent, High Reliability Distributed System."
Proceedings of the Eighth Symposium on Operating Systems Principles, December 1981, pp. 169-177.

REDE 80 Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., Purcell, S. C.
"Pilot: an Operating System for a Personal Computer."
Communications of the ACM, 23, 2 (February 1980), pp. 81-91.

RITC 74 Ritchie, D. M., Thompson, K.

"The UNIX time-sharing system"
Communications of the ACM, 17, 7 (July 1974), pp. 365-375.

STUR 80 Sturgis, H., Mitchell, J., Israel, J.
"Issues in the Design and Use of a Distributed File Server."
Operating Systems Review, 14, 3 (July 1980), pp. 55-69.

SVOB 79 Svobodova, L., Liskov, B., Clark, D.
Distributed Computer Systems: Structure and Semantics. Technical Report LCS/TR-215, Laboratory for Computer Science, M.I.T., Cambridge, Mass., March 1979.

SWIN 79 Swinehart, D., McDaniel, G., Boggs, D.
"WFS: A Simple Shared File System for a Distributed Environment."
Proceedings of the Seventh Symposium on Operating Systems Principles, December 1979, pp. 9-17.

WARD 80 Ward, S.
"TRIX: A Network-oriented Operating System."
Proceedings of COMPCON '80, San Fransisco, Feb. 1980.

WULF 74 Wulf, W., Cohen, E., Corwin, W., Jones. A., Levin, R., Pollack, F.
"Hydra: The Kernel of a Multiprocessor Operating System."
Communications of the ACM, 17, 6 (June 1974), pp. 337-345.