Using Large-Scale Operating Systems' Designs For An Eclectic Design

The MOSES Project:

Meta Operating System And Entity Shell

Daniel J. Pezely

2 February 1992

1 Introduction

MOSES, the Meta Operating System and Entity Shell, is an implementation design for intercommunication of tasks with devices, all forms of storage, and other tasks, such that the client-task has no need to distinguish between remote and local entities. Although this design was developed for specific use in virtual environment platforms, independent of detailed knowledge of pre-existing operating systems' designs, our design shares similar features with many modern large-scale operating systems.

A brief overview of the MOSES design is given, followed by descriptions of some of these other operating systems, and concluded with a summary of the MOSES design defined in terms of these other operating systems.

2 Overview of MOSES

MOSES is a modern operating system intended for large-scale use which has a minimal structure, thus allowing all internal functionality to be modified, if not replaced altogether. Our primary design concern is to minimize system downtime for upgrades and crashes which leads to minimizing obsolescence of the system as a whole even though individual components may be wholly replaced.

While the previous paragraph describes the nature of the system, here is a simple description of what the system needs to do:

Reliable communication between entities is needed. Entities are everything from computational tasks to devices, which includes communication to and from users, access to and from all forms of storage devices, and communication to and from different computer systems. And naturally, in the context of operating systems, "users" refers to both client program code and people, which at the level concerned, there is no distinction.

Although the design was originally specialized, rather than have an information engine which processes two or three dimensions of data, or even process n-dimensions, we need to move arbitrary information around, as with operating systems in general. The term *move* is used to imply that the data being stored, accessed, and communicated, are not restricted to any preset type or data structure. The system transports information to and from these entities, leaving interpretation to the entities requesting the communication, yet a single internal data structure is used for encapsulating all data.

Immediately, the question of data consistency comes to mind because not all computers store bits in the same order and not all networks carry bits in the same order and the possibility of using heterogeneous systems always exists today.

To satisfy the reality of bit consistency, the entities must register with their remote counterpart and *negotiate* upon which formats to use, if other than the default formats. By negotiation, we mean specify which protocol to use. This negotiation frees the communicating entities to use whichever protocol is optimal for their conversation, and as better protocols are implemented, they may be used.

To expand the range which our entities may be: everything is an entity, and all entities may be communicated. This means the protocol evaluators are entities, and if the remote and local entities can negotiate upon a protocol

for exchange, the protocol evaluator code may be transmitted to another entity (in this case, system). This makes use of dynamic linking, and since the system is intended to make remote-versus-local distinctions transparent, we also have remote dynamic linking.[1]

Since there is some discrepancy between researchers about the exact meaning of "dynamic linking," we are referring not to load-time linking or run-time overlaying but the loading and linking of modules which may have been created after the system has begun running, without the need to restart the system.

From this brief overview of requirements, it should be clear that there is no new theory required for the design of this system, so applying existing theories and systems' technologies should be sufficient.

3 Previous Systems

A brief description of pre-existing systems and platforms is presented from the perspective of our design as if our design was modeled after that particular design.

Complete descriptions of each system described and elaborations of benefits of particular features are not presented in this document; instead the bibliography lists works related to our design and the designs presented here. Prior knowledge of these designs is not required to understand our design.

Some features are not mentioned here which may have been elaborated upon in the original documents, and since some features were novel then but now assumed, such descriptions are left for the reader to find in the original documents. Examples of this are: memory management services making use of the hardware-provided memory paging and protection features and having sufficient local memory to provide caching for just about everything.

Following all of the design description, the MOSES features which differ or extend the designs are given.

4 Prospero: The Virtual System Model

The project developed by B. Clifford Neuman at the University of Washington called *Prospero*, an implementation of the *virtual system model*, creates

a user-centric view of a global file system. The model provides more functionality by being included as part of another operating system rather than by being a stand-alone system itself.

For an abundance of technical reports on Prospero, the virtual system model, and associated proposals and research, see Neuman's papers [3] [4] [5] [**?**] [7] [8].

The file system service provided spans to include any file on any file server on the network, theoretically anywhere in the world. The name space is user-centric in that the user controls what is named in that space.

The usefulness of the virtual system model is in what it hides from the user. The claim is that with a global file system, there is far too much information accessible which can easily overwhelm the user. By limiting the volume of information that is immediately shown to the user, with the remaining hoards still accessible upon request, the user sees small digestible portions of the total environment, which in this case is presented as a file system.

The user-centric name space for finding files in this global file system is completely under the control of the user. The user not only can control which directories from which hosts to include in his/her/its domain, but individual files to be included are also at the discretion of the user.[1]

A step to further increase user-control is the use of filters. A filter is a task associated with a link to a file. The virtual system model explicitly specifies that this task take the form of an executable program, but clearly the design includes the possibility of kernel's internal functions to be dropped in as easily as programs for small file systems residing in main memory.

User-centric name spaces, union links, and filters keep the user's view of the world limited only to that which is of interest to this particular user.

It is crucial to the design that the filters be associated to links and not to directories. That is, filters provide the interface on a one-to-one relationship with the files and not one filter to many files as would be the case with interfacing only to directories.

Different users should be able to, of course, access other users' names spaces, so any implementation should handle access rights. Filters may handle such access rights which then would put such matters entirely in the

[1] Selecting files from multiple directories to be accessed as if they all resided within the same directory is referred to as *union links*.

hands of the users.

The virtual system model is not just for human users to find their way around a global file system. The model provides the scoping and a controllable environment for applications and services as well. And again, one element of the control is security.

Scoping allows applications to run consistently for different users because everything the application needs is known through an environment (the name space) which is associated with that application. Continuing the idea of scope, security of access rights may be handled as a function of the environment. This functionality may be implemented as a filter function, and these environments are what give users their view of the global system.

The concept of an authority is used in the virtual system model, implemented as a remote name space. An authority is anyone who knows more about a particular entity than you do, including knowledge of another entity's existence. When information or resources are being sought, an authority may be asked to share its information. This is how additional links to remote systems may be found.

The presentation of the model thus far is quite vague on one point: the exact form of information to be stored and accessed.

The storage from within the model is presented as a file system in the Prospero documentation but by abstracting the functionality of a file system and imagining uses of filters, any form of memory should be accessible as different name-spaces: a task's local memory, a system's main memory, local disks, remote file servers, etc.

Access through filters and environments includes access to other system resources, such as processors; thus, there is more functionality to the system than just the global file system aspects. The important point here is that as an operating system, the resources are accessed using the same abstraction as with the file system. However, this is not limiting access to other devices, just providing a common interface.

Whether accessing a remote file or another remote resource, the remote system still has control of that resource, which provides a base level of data security/integrity. And by using filter functions to reach remote systems' resources, local systems have drop-in security against local users accessing remote entities.

Some issues raised by Neuman are closure, authentication, and sharing data with naming conflicts.

Closure is the need for association of remote links to local data to have some identification of the remote entity linking to that data.

Authentication is necesary whenever remote requests are made. Knowing and verifying who is making requests before granting those requests in a reasonable amount of time is very important. The importance increases as the systems are used outside of a research environment.

Sharing data with naming conflicts between users always exists. Since multiple users may have the same name for different entities, a sense of local versus remote naming must be introduced such that anything outside of the user's namespace is named according to which other user's namespace the entity resides. Any concept of an authority may be viewed as simply yet another user with its own name-space.

5 Amobea: A Capability-Based System

The elements of *capability-based* operating systems which are examined are the protocols, communication mechanism, file system, process management, and resource management, avoiding detailed issues of security and authentication.

The other features of the implementation model are not important for our discussion. See [2] and [9] for a full description of the model and *Amoeba*.

S.J. Mullender and A.S. Tanenbaum describe a system design using capabilities called Amoeba. This system was described as being radically different from existing systems, in their 1986 paper. The difference was due to a rejection of rigid multilayer protocol stacks which introduce extra levels of indirection thus makes the system inefficient.

Using the design philosophy of keeping the system kernel as small as possible and using abstract data types for controlling/accessing system resources allows a more efficient implementation which has many benefits. The two primary benefits are as expected: a higher level of user abstraction for dealing with the systems and its resources, and without rigid protocol stacks having to be implemented within the kernel, such stacks may be introduced later, thus moving one cause of obsolescence from the kernel to outer system layers.

Also, in the same manner which the protocol stacks are optional to the system and may be external to the kernel, so too, features like security and

authentication are optional to the system and may be external to the kernel.

As is the trend with modern systems, connection oriented communications are being avoided as the default message passing mechanism. Instead, messages are passed as *transactions* (to use Mullender's term) which take the form of a small hierarchy of packets based upon datagrams. Note that this hierarchy is not a strict protocol stack but users will typically access just the top level. The levels deal with degrees of reliable transmission and network access, as expected.

By having tight control over the protocol levels, optimization of packettizing messages is possible. Mullender reports of two-thirds of file-system accesses can fit into single packets of 2K bytes. Thus packets can allow for a "request-reply" or "transaction" protocol implementation.

Services of the system are accessed through ports where communication actually takes place. These ports are known only to the server and each client, and potential clients for common services will generally know of such a service and its ports. Knowledge of a port does not mean service will be granted automatically.

Security and Authentication of service users may be handled by hardware or software external to the kernel. Being external, such features do not expand the size of the kernel and may be replaced without modifying the kernel, thus lessens the reasons to upgrade the kernel, thus lessens the number of system shutdowns.

Getting into more traditional operating system functionality, capability management is distributed, as expected for a modern system, not centralized. Capabilities may be viewed as the protocol of the core operations of the system which replace system call access to kernel features. That is, access and manipulation of system resources such as the file system are controlled through this protocol, or *capabilities*.

The syntax of a capability is such that some authentication is made. Other syntactic features are for the server port number, entity within the server such as an internal function call, and access rights.

Common services are those of file systems and process management. First, the file system is presented. The file system consists of three primary services itself. The File service deals with linear files which permits those files to have internal addressing (relative to itself) and does not fragment the files. The Block service deals with raw device access and has no concept of files or file systems. The Directory service deals with symbolic names of files

to locate internal file reference values.

Processes, which may be seen as accessing the system resource pool of software applications, are managed in this design by three subsystems: the Generic server, the Process server, and the Boot server.

The Generic server will handle common applications, and the example given by Mullender is that of a pascal compiler targeting a MC68000 architecture. In this case, a cross-compiler may be used on, say, a VAX to generate the target code. The user will not care what system actually runs the compiler, provided the target code is correct. So, the Generic server handles such cases of using common software applications.

The Process server is used when there is no support from the Generic server, such as when that user wants to execute the compiled Pascal program. Since there will be no knowledge of this program to the Generic server, the Process server will initiate a new process/thread to handle the execution of this program.

The Process server must be informed of, among other things, the capabilities of the process/thread to be executed. For most current operating systems, this refers to file descriptors and environment variables.

The Boot service keeps other services alive. By periodically checking the status of services, including its own subservices which check the Boot server itself, should any of these services crash, the Boot server will restart that service. The nature of the services dealt with are things such as the file, directory, and block services which support the file system.

6 The V Kernel

David R. Cheriton's V distributed system at Stanford University was developed with the design philosophy that the communications protocols, not the software, define the system, and the other tenet is that this protocol must provide high performance because the most elegantly designed protocol is worthless if the system is not usable. It is the philosophy of the design which is focused upon here.

For a description of the V system with details about the implementation, refer to Cheriton's 1988 CACM paper [1].

The kernel is described as a software backplane, functioning like the backplane of a computer bus. The functionality is a network-transparent abstrac-

tion of address spaces, lightweight processes, and interprocess communication.

Interprocess communication is performed via a fast transport level service for small messages such as remote procedure calls. The kernel handles remote and local cases of message transport differently to optimize local routing. For remote routing of messages, the VMTP transport protocol is used which is a request-response protocol.

The protocol supports multicast communication through an additional layer of indirection, which may be seen as an alias: a single pseudo-entity being addressed to which does the multi-destination message sending. The V kernel does this aliasing by logically addressing a process group rather than an individual process, but the abstraction still holds.

The intercommunication mechanism provided by the kernel allows kernel services to effectively be applications of the kernel although may reside within the protection of kernel memory domains. Therefore, interprocess communication between a user process and a kernel service is identical to communication between user processes, as is identical to most communication between the kernel and its services. Using a message-passing facility for communication rather than system calls permits the mechanism to be modified independently of the service modules and without the need for the system libraries being modified.

The system is packaged with kernel plus all of the basic services to provide the minimal functionality of the system, such that each server only has to deal with the case of local requests, thus simplifying the implementation of each server. The benefit is that all basic requests of a kernel service will be granted by intercommunication with a local service, reducing the implementation complexity of those services.

This design of kernel service communication also allows additional system services to be dropped in cleanly without recompiling the kernel.

Some system services include a real time service synchronized via the interprocess communication facility, thus avoiding the need for a dedicated or complex time protocol yet sufficient enough for real-time applications.

Process management services for creation and destruction of a process is separate from that of memory allocation services, specifically for creating the address space of the process.

File descriptor management services are handled separately such that overhead for destruction of a process is minimized, thus a separate garbage

collection service must monitor which processes are no longer existing and clean up the file descriptor tables and such accordingly.

Scheduling services within the kernel are minimized in complexity using a simple priority-based scheduler, and more complex scheduling is handled as an external service, any theoritically may be replaced at run-time as system loads change.

Memory management services load a program file on demand such that there is no special mechanism for program loading versus mapping a file into an address space, and file-like access to address spaces are permitted so standard user input/output routines may operate on any file or address space, providing a common interface.

Device management services handle access to all devices while being device-independent and hardware-independent yet still frees the kernel from the associated tasks with the exception of the lowest-level interrupt routines which must be privileged for kernel integrity.

User input/output facilities are library routines which provide a higher level of abstraction than individual messages, thus consistent with other operating systems' interfaces. The differences with other systems start with that I/O is block-oriented, not byte-oriented, thus provides an efficient transfer of data to entities which can accept multibyte increments of data. Secondly, I/O is *stateful* which is useful for locking and recovery with remote links to data. Finally, there are three types of interface functionality: compulsory, optional, and exceptional which respectively define the base-level functionality, additional functionality, and escaped functionality for specialized operations.

When it is necessary to find a file or a resource using a name, the V design has each manager implement the naming for its own set of objects; thus, there is no need for a master naming service for everything within an object. However, a master naming service may be used to find other services, and each process may have a local cache of the services' names it uses.

Names to objects within remote services are identified via string names with prefixes which correspond to each particular service. This naming scheme creates a shared memory. Names are different from object identifiers since the overhead associated with string-names should be minimized. Object identifiers are fixed size bit strings with fields specifying manager/server and local object within that server.

Object identifiers are used only as non-static pointers, thus must be reassigned after a crash and restart, unlike names which may still be valid after

such cases. In a deeper level of identification, entity identifiers are also fixed size bit strings but without internal fields defined.

Entity identifiers are host-address independent, thus migrateable, but this causes a conflict between reasonably sized identifiers and the problems associated with reuse and expected identified entities being seemingly altered.

7 MOSES, As An Eclectic Design

The MOSES design starts with the concept of an *entity* which is a generalization of memory (files, processes' memory spaces, and memory-mapped resources), functions (processors, processes/threads, tasks, and system call libraries), and communication (protocols and remote resources, services, processors, libraries, authorities, etc).

Keeping with the V design philosophy, the key of the system design is the protocol, and performance matters for elegant designs in the context of large-scale operating systems. As agreed upon by the designs considered, a request-reply protocol is used for message-passing. Since most messages/requests will be small in size, packet headers are optimized for such cases while local delivery cases are treated special. Second of the design criteria is network transparency, as is typical of most modern systems.

To satisfy the needs of the entity generalization and to support the desired protocol features, an abstract data type is used internally to our system, as with the systems reviewed. The data type which is used reflects the internal structure of a file system to provide data integrity while allowing concurrent accesses; however, this form of security is performed block-by-block and not for whole files. This block-by-block form of security corresponds directly to Amoeba.

An additional feature which does not seem to be in any of the designs studied but could possibly be added simply, is that the single data structure used by MOSES allows nesting and recursive links and doubling up of data blocks which eliminates the need to duplicate data just for the sake of having a copy. Due to this feature of effectively nesting tuples, the term we use to describe our structure is *grouple*.

Keep the kernel as small as possible: more sound advice from our predecessors. Small kernels simplify real-time versions of the system, if necessary, and in the applications intended for our system, real-time is definitely on

the list of future projects. Also, by keeping the kernel small, there is less to break, thus less to fix, thus less to worry about, thus hopefully translates to minimizing downtime and upgrading.

To provide for network transparency and giving users the freedom to access both local and remote entities with a common address format can be done a number of ways, as demonstrated by the systems described in this document.

Our system provides an address structure as part of our grouple structure which handles both internal/local addressing and external/remote addressing. The remote addresses are defined in terms of the user's local name-space (*DataSpace*) and have corresponding closure-entries pointing back to the user from the remote entity. The remote address is maintained in place of would-be local data and linked to by a Remote-DataSpace service which keeps track of all remote references. Likewise, on the remote end, there is a corresponding service keeping track of links to local data by remote addresses.

The Remote-DataSpace service provides the closure which the virtual system model brings our attention to, thus cases of amnesia are recoverable and avoids dangling references, all without the overhead of adding closure information to each data block structure.

To handle the issue of data integrity, especially with remote links, version identifiers will be maintained within addresses and the headers of the actual data. This corresponds to each of the systems reviewed.

Independent of network topology, MOSES routes messages entity-to-entity rather than just port-at-host to port-at-host. This form of message routing is similar to that in the V kernel, which is also network independent. Both Amoeba and V optimize routing for local and for remote cases rather than use a single generalized case, and we agree with this decision.

Routing entity-to-entity is inherently a non-multicast system, but multicasting can be performed indirectly through an alias which is a single entity containing the addresses of all target entities for the multicast. Again, this corresponds to many current systems and notably, V.

Routing tables are maintained by a one-time start-up registration. Changes to registration information are done through either sending another start-up registration and thereby invalidating any corresponding existing information or by sending a re-registration message with just the changes. The start-up registration is typically sent after a new installation or a total system crash and cold boot. The emphasis put on crashed systems agrees with the V

system's expectations of reality.

The protocol used by MOSES is nothing more than a transport mechanism, just like with the other modern systems. But, from the ground up, the protocol provides access to the physical network, support for request-reply messages, multi-packet single messages, and multiple sequenced messages. All layers are user accessible, all of a configurable reliability, configured by entity-to-entity registration and/or by what is effectively system-to-system or system-to-gateway registration.

The V system only allows user access to the top level of their protocol stack with degrees of reliability increasing towards the user level. Prospero has a similar transport mechanism to MOSES, though as currently implemented, not as configurable. In each system, rigid protocol stacks are avoided in the core of the design, but may be added later, externally to the kernel.

By having a one-time start-up registration, the registration information is effectively cached by both sides of a registration, and again, this agrees with the Prospero, Amoeba, and V.

Message-passing between remote systems is performed via randomly selected ports. This provides minimal security by way of hiding information, but just as additional protocol stacks may be added (explained below), so too security such as encryption may be added later, externally to the kernel.

Security and selecting an alternate protocol stack is handled via late binding of protocol. This is done as a function of the grouple data structure within MOSES, and similar features exist in each of the systems reviewed. As with filters in Prospero or by replacing the packaged services which are external to the Amoeba and V kernels, alternative protocols offering security may be dropped into MOSES either as an external process or linked internally within the kernel.

This brings up an important feature of MOSES which could only be added to the other systems after a major rewrite effort.

Additional functionality may be installed as an external process or as an internal function routine within the kernel. Without discussing the internals too deeply, the mechanism is addressing the scheduler versus addressing a function pointer, but the mechanism is designed to provide maximum flexibility of the kernel. None of the systems reviewed have this feature.

Through this user of changing internal functions and through the use of dynamic linking—referring not to load-time linking or run-time overlaying but to linking and loading of modules possibly created after the system has

been started up—internal functionality of the MOSES kernel may be wholly replaced on-the-fly.

Upgrading on-the-fly is then possible from a (trusted) remote service, without the need to restart the system, thus saving all volatile memory, while only producing a one-time delay when viewed by other systems and services.

Some service packages which may reside internally or externally are the scheduler, the spawn service, the router, the message-delivery facility, the memory allocation server, file system services, and the boot server.

Each service is fairly common, thus expected, but the boot server is the same as with Amoeba's boot server and is solely for the purpose of keeping other services alive with subservers to keep itself alive. Also, the memory allocation server has subservices for internal/local processes' memory and subservices for external/remote device memory such as on disks.

For a native MOSES file system, directories are inherently treated separately from the files which are also inherently different from the individual grouples (blocks) which compose the files, and this corresponds directly with the file system family of services within Amoeba.

Again as with Amoeba, the process services provide a "generic" service which maps process creation requests for commonly used processes into registered services. Such a re-mapping service could easily be implemented within MOSES, but at least initially, the task of first checking registered services resides with the user to check the registration tables.

Getting back to the dataspace concept, rather than rely upon just existing dataspaces as with Prospero, and rather than creating a de facto shared memory space, a global memory system is created, called the *DataSpace*. This dataspace gives an entity- or user-centric view of the system when applied to a local system, and that one node (call it a subspace) is part of the global DataSpace when viewed as a distributed system.

Remote dataspaces are addressed just as any other remote entity. Initial entity access may be via looking up symbolic names in another space or by requesting a name from an authority, but both are implemented ultimately as another DataSpace.

Depending upon how a different dataspace is accessed as a subspace, scoping and authoritative spaces may be set up. This matches the scoping efforts of Prospero.

Symbolic names are represented by scalable multibyte characters; that is,

for each string of symbols defining a name, each character must be the same number of bytes, but that number may vary from string to string. Without the scalability at run-time, opposed to at compile-time, this is similar to "entity identifiers" within V.

Such symbols are the foundation of the addressing scheme mentioned earlier. By allowing scalable multibyte symbols, transportation and delivery of multibyte character string messages can be optimized for local delivery and remote delivery.

Also, scalable symbols allow optimizing internal messages to the local processor and to remote processors. This of course applies to all local and remote resources as well. And, such symbol strings work well for Asian languages which use multibyte characters.

The MOSES system is packaged with kernel plus all of the basic services to provide the minimal functionality of the system, even if all a service does is nothing more than itself make requests of a remote service. The benefit is that all basic requests of a kernel service will be granted by intercommunication with a local service, which then may make a remote request.

To respond to the issue of implementation complexity and the significant performance loss incurred should such a feature be used, which is the reason the V kernel does not do this, this provides for a means of running with the latest system kernel service features temporarially without having the upgrade.

However, for efficiency's sake, the criteria with the V kernel for having the full kernel services wholly implemented with each system package is highly desirable and will remain our default case, but we wish to not exclude any possibilities of functionality.

8 Conclusion

It should be quite clear that there are no new theories within the MOSES design; however, there are also no existing system implementations which have all the features which we desire. But should there exist such a system, we will still finish our implementation just so we can have the fun and excitement of writing our very own *yet another distributed operating system.*

---

9 Addendum

10 February 1992

One system in particular which *seems* to have all of the MOSES features is the ISIS distributed system from Cornell University.

This system has just been brought to my attention, and a future version of this document will describe ISIS.

References

[1] David R. Cheriton. The V distributed system. *Communication of the ACM*, 31(3), pp. 314-333, March 1988.

[2] S.J. Mullender and Andrew S. Tanenbaum. "The design of a capability-based distributed operating system," *The Computer Journal*, vol. 29, no. 4, pp. 289-299, 1986.

[3] B. Clifford Neuman. "The need for closure in large distributed systems," *Operating Systems Review*, vol. 23, no. 4, pp. 29-30, October 1989.

[4] B. Clifford Neuman. The virtual system model for large distributed operating systems. Technical Report 89-01-07, Dept of Computer Science and Engineering, U of Washington, Seattle, WA 98195, April 1989.

[5] B. Clifford Neuman. Workstations and the virtual system model. In *Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems*, pages 91-95, September 1989. Also appears in the Newsletter of the IEEE Technical Committee on Operating Systems, vol. 3, no. 3, Fall 1989.

[6] B. Clifford Neuman. The virtual system model: A scalable approach to organizing large systems, a thesis proposal. Technical Report 90-05-01, Dept of Computer Science and Engineering, U of Washington, Seattle, WA 98195, May 1990.

[7] B. Clifford Neuman. Proxy-based authorization and accrounting for distributed systems. Technical Report 91-02-01, Department of Computer

Science and Engineering, University of Washington, Seattle, WA 98195, March 1991.

[8] B. Clifford Neuman. "Advances in Distributed Computing: Concepts and Design," chapter within *Scale in Distributed Systems*. IEEE Computer Society Press, 1992.

[9] Tanenbaum, A.S.; Renese, R. van; Stavern, H. van; Sharp, G.J.; Mullender, S.J.; Jansen, J.; Rossum, G. van. "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM*, vol. 33, no. 12, December 1990, pp. 46-63.