MOSES Kernel Implementation Design

The MOSES Project:

Meta Operating System And Entity Shell

Daniel J. Pezely

11 May 1991

1 Introduction

We needed a very flexible operating environment suitable for research, design, and development of applications—applications for wide-area multiple users desiring more than just control but requiring participation with the data and functions within the application. This environment requires a solid foundation to run upon, and this paper describes that foundation.

Readers from outside distributed systems development should be able to follow the abstractions and design arguments, but our primary goal here is communicating to operating and distributed systems people. Other papers give informal specifications of the design and others give formal specifications of individual elements, such as the communications protocol suite.

We start with a brief list of other research areas which are applicable to this project and a short mentioning of some design features from those areas. Then, we present a quick overview to give distributed systems researchers a frame of reference before explaining the software architecture. A summary is given in this paper, but analysis and applications are described in companion papers. [1]

2 Background

From an operating systems perspective, we looked to UNIX [2] 4.3BSD, Linda, Mach, Plan-9, and Amoeba. See [3], [4], [5], [6], and [7], respectfully, for their designs and results.

The UNIX operating system allows pipelining processes together so that one program's output is another's input. Pipelining in this case is more than just another name for interprocess communication (IPC), but the ability to interpret an IPC stream as the standard input or standard output data stream has benefits in itself.

These benefits provide very powerful techniques which our design should possess. By using such tools, a uniform access to environment resources may be achieved through simple tool hierarchies.

In short, we derived our communications design from the knowledge and experience gained from the Internet Protocol (IP) suite [3] [10] and the Open Systems Interconnection (OSI) protocol stack [4] [11]. We have reviewed the design descriptions, analysis, and commentary of designers, implementors, and experts in the networking field. [5]

The IP flexibility to bypass one layer and access an underlying layer is very desirable yet something which cannot be done with OSI.

The components used from each system are discussed further below.

3 Software Architecture

In short, MOSES is a distributed, message-passing, open operating system providing system independence and not just hardware independence.

[1]See [1] for analysis and [2] for developing applications.
[2]UNIX is a Registered Trademark of AT&T Bell Labs.
[3]The DoD ARPANET model.
[4]The International Standards Organization's *Open Systems Interconnection* model.
[5]References include Dave Mills <mills@udel.edu> and [9] [10] [11] [8]

3.1 Design Overview

MOSES stands for *Meta Operating System and Entity Shell.* A meta operating system (meta-OS) is a layer above the native operating system (OS) but below applications. Such a layer fits into, but is not restricted to, the OSI Presentation layer. (See Figure 1.) Our design also covers the OSI Session and Application layers, but like the Internet model, the Process/Application Layer may be bypassed completely or built upon using any number of application-supplied interface layers. (See [11] for an overview of these and other protocol layers.)

Our design gives users access to underlying (native) OS resources. Additional features provide for multiple users, distributed shared memories, and parallel execution, across internetworks.

3.2 Special Characteristics

The Entity: any and all arbitrary information which can be described symbolically. An entity's inherent components consist of memory, function, and communication, but the boundaries between these elements blur since memory and communication may be seen as functions, and communication may be attained through memory/storage. [12]

Entities are more abstract than programming language objects; since everything is an entity, everything looks identical to the system. This apparent identity allows any feature or user to be anywhere in the system. Location possibilities are limited only by the extent of the networking and distributed systems.

The DataSpace: unity of all memory which appears to be under the exclusive ownership and complete control of each user while being a distributed, shared, associative memory with internal and external addressing permitted. [13] (See Figure 2 for the user's illusion and Figure 3 for the actual DataSpace organization.)

To provide simple distributed systems solutions, primitives and segmentation make DataSpace usage transparent to users. Other shared memories may be implement on top of the DataSpace since the DataSpace only defines the control primitives and handles apparent boundary-less segmentation.

3.3 The Kernel is the User Shell

One key architectural design element makes the core program in this design equivalent to applications, but more, the core program entity may be called a kernel *and* a shell.

We often refer our system as a kernel intentionally for the comparison to an OS. Although our design does not specify any low-level operations which might be expected in a kernel, all the features *could* be implemented. Rather then re-implement an entire operating system for each hardware platform, we build upon existing completed systems and keep our kernel as an application of other OS's.

3.3.1 Kernel Services and Features

Like other message-passing OS's, design extensions and additional features may be handled by implementing kernel services (daemons). So, unlike the original UNIX kernel which has grown to include many features, this kernel should remain very small and all additional features will be implemented externally. In fact, the DataSpace primitives require only a daemon; all users of the DataSpace shall access *stem* routines which communicate with the external daemon.

3.3.2 Communication Protocol Suite

And we often refer to our system as a shell since the protocol suite shall be accessible to the user directly. That is, the user may talk to the system through any protocol in the suite via tools or having typed keystrokes becoming a data stream for a protocol interpreter. Such accessibility of all the protocols makes all users of the system—humans, applications, and daemons alike—identical. To provide for human interaction, very high level protocols may be used, and the Common Lisp programming language is one early shell protocol used.

Communication between non-human users, of course, make use of more efficient protocols than text-based programming languages.

3.4 Data Structure Overview

There is only one pair data structures in our design and they serve one purpose: storing lists of information. We omit the actual data structure definitions in this paper since we are emphasizing implementation strategies and not implementation specifications.

Inside the kernel, we only distinguish between basic data types to prevent data corruption. The data types used internally are symbols, pointers to nestable lists, and pointers to functions.

3.4.1 Physical Storage Locations

Data can be stored in any location which the native OS can access (disk, RAM, special devices, etc), but the data structure must provide for a uniform access. That is, by accessing information in the database, if that entity's data is actually on some remote or external device, we must be able to access it easily and cleanly. We allow access through functions which are stored in the database. The functions are not the data, but the value the functions produce through computation or through remote-procedure-calls (RPC's) produces the actual value.

3.4.2 Use of File System Design

We are providing a shared memory, and our design draws from existing designs. The BSD fast file system, a simple shared memory, handles small blocks of information well and the design is sufficient for our needs at this time. [3] Amoeba's Bullet servers are designed to be faster for storing complete files. [7] However, the fast file system better suits our requirements which are similar only to storing inodes and blocks and not to storing lengthy files.

Networked file-systems used heavily today have demonstrated the effectiveness of shared file system designs. However, the key to managing network file systems is having a bottleneck: the file server. This limitation simplifies resource sharing and mutual exclusion of data, thus systems become dedicated to the sole task of file system service. We accept this design for our system and are evaluating this approach from the standpoint of tomorrow's systems. Naturally, research will continue after we analyze performance of this design.

3.4.3 Use of Lisp S-Expression Design

Since our kernel makes no distinction between different types of information, we looked to various list processing techniques. The lists to be managed are arbitrary. That is, there is not structure or length imposed upon the lists. Much of the list data structure borrows from Lisp s-expressions [14] but is not limited to that.

3.5 Grouples and Symbols

The name we assigned our version of information lists is *grouple*. Unlike s-expressions in Lisp, the user stores all information for distinguishing between various atomic data types in a grouple and references the data as a nested grouple. A grouple stores bits and treats bits of atomic elements as generic *symbols*. Non-atomic elements like data types and property lists are maintained as regular lists of lists or lists of symbols and accessible to the user directly or through functions.

Characters, integers, and floating-point numbers are interpreted as such at the points of use. Symbols are passed to routines or external users, and they determine what the symbol data type is. This removes any restrictions on the precision of numbers and allows heterogeneous systems to work together.

3.5.1 Grouples as Lists

One grouple contains one list of information elements. Since grouples can be nested, some information elements may reference other grouples to form the nesting. Each grouple is typically small and may be chained to form hierarchies. This chaining is similar to data blocks referenced from *inodes* in UNIX file-systems, hence another reason for using a file-system-like storage strategy.

3.5.2 Grouples Store Entities

An entity is one grouple, but this "grouple" includes all its nested grouples and includes all of their nested grouples and so on, until symbols are reached. We view symbols as leaf-nodes in a tree but do not restrict ourselves to only trees. Although the design permits nesting of complete grouples, the recursive concept of any one entity being one complete grouple and each of its children being complete grouples themselves, is very important. The benefits due to this recursion simplifies implementations.

The chain of all grouples allocated within a single system form a unity called the DataSpace.

4 Entity Memory—The DataSpace

To provide the functional unity of memory and the uniform access of information which has been mentioned earlier, we use the DataSpace. The DataSpace functions as an associative memory similarly to the tuple space in the LINDA model. [4] What LINDA omits from the specifications, namely the structure of the tuple space, the DataSpace provides. Our structure chains of all allocated grouples (nestable lists of arbitrary information) together which in turn may be treated as a whole. That is, we may treat the collection of all stored entities as a whole.

This structure assists for debugging the DataSpace, best understood with a disk-file analogy. With a disk file, we may view data blocks as a single object: the file. With the DataSpace, we may view all grouples as a single object: the DataSpace. Likewise, the entire DataSpace may be examined from start to end, just as a plain disk file may be examined. Design of a debugging-entity follows easily from this feature.

4.1 Segmenting the DataSpace

By segmenting the DataSpace, it can be distributed. Additionally, we reduce searching complexity for the DataSpace using indices, as is common in relational databases; however, *subspaces* used as indices are part of the DataSpace rather than being distinct data files themselves. Some of the standard segments, or subspaces, within the DataSpace are: the SysLibSpace and the UsrLibSpace. The parser imposes these subspace classifications, and as described in the parse section, the user completely controls all aspects of the parser.

Keeping things simple and consistent and even recursive, each subspace is a space in the sense that the DataSpace is a space. Inside the kernel, the same pointer type which references the DataSpace is the type of pointer which references each subspace. This gives us uniform storage facility access and uniform access both internally to the kernel and externally for the user.

A subspace, then, is a list of grouples contained in that space. The kernel's internal subspace pointer used references that grouple, and the DataSpace's internal pointer references the first grouple in the DataSpace. Again, we chain all grouples together in the DataSpace; therefore, from the first grouple, we can access the entire DataSpace.

4.2 Subspaces, Described

The subspaces, which may be called spaces, are entities and are represented using grouples. Therefore, a space may be local or remote, structured or unstructured, a tree or a table, etc. Since we may use a grouple element as a symbol, a function, or reference to another grouple, we may develop various types of spaces. The DataSpace management routines and the parser routines will act accordingly for the different types of spaces. (The routines and modules mentioned in this section are detailed in their respective sections of this paper.)

4.2.1 WorkSpace

The WorkSpace holds all input and output messages awaiting service. The main kernel function operates over the WorkSpace, and this space is known explicitly only by the kernel main loop. This space is used as a temporary area for incoming messages and partially resolved (parsed) grouples. Once resolved, the parser either initiates that grouple's equivalent routine or stores the grouple in another space.

By keeping the kernel routines generic enough, they may work with any subspace or the DataSpace as a whole.

4.2.2 SysLibSpace

The SysLibSpace contains the built-in system routines (the command library). These routines provide user functionality. Like any other space, the SysLibSpace may have any structure for further segmentation, thus simplifying parsing.

4.2.3 UsrLibSpace

The UsrLibSpace contains all information, variables, and functions, defined by the user [6] These definitions are created by input messages, thus may originate from local users, remote users, script files, remote servers, etc.

4.2.4 SpaceSearchOrder

A grouple naming which other grouples to search and in what order is convenient, hence the SpaceSearchOrder grouple/space. In general, the UsrLibSpace precedes the SysLibSpace, allowing users redefinition of system routines and variables. This space may be viewed as a *path* used in many OS shells or as the scope order and extent in Lisp.

4.2.5 Other Subspaces

We often compare the SysLibSpace and the UsrLibSpace relationship to privileged versus non-privilege directories in UNIX for access characteristics only. The parser assigns names to these and other spaces upon start-up configuration.

The communications library maintains routing information and allocated communications channels with spaces. We highly encouraged all system services to follow this example and store all internal information in the DataSpace. The information, like any other information in the DataSpace, would be stored as symbols whose data type is known internally to the service. This symbol usage allows users access to the data while maintaining internal data type efficiency.

4.3 Accessing Spaces

As mentioned above, we may structure any space via hierarchies, functions, or flat symbols. When functions are used, the parser accesses the spaces through a call to the given function. This function call, specified by the grouple representing the space, may be a remote procedure call (RPC), thus we may access remote spaces. Simpler than accessing remote spaces, we may access non-spaces such as programs and functions.

[6] Again, the *user* may be human or an application.

5 Entity Function

Here, we speak of "function" to mean all executable routines. The system accesses local routines internally through function pointers, but the user gains access externally through interface routines known by the parser. By using interface routines, the actual function could be local, remote, built-in, or a process to be spawned. Using standard remote procedure call (RPC) terminology, the interface routines are the stem procedures for remote routines.

5.1 Core Routines and Genuine Dynamic Linking

To maximize kernel flexibility, only a minimal set of core routines are built-in. The remaining routines, including the entire user command library, are loaded at run-time via dynamic linking. Genuine dynamic linking [15] allows loading compiled program objects from disk while the program is executing; however, unlike with just shared-libraries, these object files may be removed. The native OS considers these dynamic objects to be data and not code. Such dynamic linking allows upgrading all but a minimal core routines without restarting the kernel.

The system daemons and applications may be loaded into the system using dynamic linking for run-time efficiency over interpreter-based linking. Dynamic linking also gives any entities access to the native OS system calls.

6 Entity Communication

Communication refers to message passing as well as accessing routines. As we all know, making programming language system calls is one communication form.

To provide consistency for the user, all communication should be identical. This is the reason for mentioning communication as a entity's component even though it is one specific function class.

6.1 Protocols as Tools

Each layer of the communication stack [7] may talk with any other layer of the stack directly and without accessing any intermediate layers. Each layer then becomes a toolbox which may or may not be used. Simple tools build upon other tools, and complex tools function stand-alone.

Using protocols as toolboxes provides as much flexibility as possible for communication end-points. By making tools out of the stack, we may bypass any layer whose functionality we do not require. More functionality may be added to the tools handling the commonly used procedures which are implemented repeatedly every day.

6.2 Communicating *Entities*

There is a notion of an entity, not of different types of entities; and again, everything is an entity: including users and routines alike. Accessing entities, then, translate to the Lisp-style of function calls. That is, accessing a function via its name or via a function pointer is done the same way in Lisp, thus simplifying things for the user. The simplicity may be thought of as sending email and accessing programs being identical. And the simplicity extends to kernels accessing a common network calling remote entities, thus forming a distributed system.

7 The Entity Shell: A Base Application

Everything the paper describes to this point may be implemented with kernel library routines. We provide the user with the full kernel library routines through the Entity Shell command library. The Entity Shell provides the user access to all the entities known by the system. To communicate with these entities, we provide a high-level programming language as the highest-level user-protocol. The protocols control the shell, the host, the communication interfaces, and possibly other hosts running a similar shell.

Although having a programmable user shell is commonplace with many of today's systems and environments, we mention it here from the kernel source code stand-point. The Entity Shell is the main loop of the kernel

[7]Protocol stacks, such as the OSI model.

executable. The default main loop selects the next input channel to read from, reads input from that channel, selects the next unresolved message in the WorkSpace, attempts to evaluate that message, and repeats these operations in a loop.

By modifying the kernel source code, programmers may simply replace the main source file with their own, and all of our modules for communication, parsing, data storage, etc, will be available as tools in a programmer's library.

---

8 Design Summary

The key of the MOSES Design is not necessarily the implementation but how distributed operating environments are used. Therefore, we may be able to impose our design upon existing platforms *or* implement everything from scratch.

The two fundamental design concepts are the notion of an entity and the DataSpace organization and structure. An entity may be any arbitrary information which can be represented symbolically, and the DataSpace is the storage for all entities. Entities have three basic components: memory, function, and communication. Again, memory/storage is handled by the DataSpace, which is also an entity in itself. Functions are entities containing information about executing a task which may access any native OS system call. Communication is the most commonly used function, but the boundary between function and communication blurs even more since, by storing information, other entities may potentially access that information, and that is communication.

The DataSpace organization and structure gives the user an illusion of exclusive ownership, but the DataSpace is a shared memory. Also, the DataSpace appears to unite all stored entities which may actually be distributed. A DataSpace storage location may seem to be just that—a location—but in fact be a function which transports information to a remote system. And the combined localities of the DataSpace may be viewed as a single DataSpace with multiple internal access points. The user transparently accesses the combined DataSpace, and the user may be human or an application.

Our system is very much user-centric. That is, the design focuses on flexibility and ease of use while providing for and allowing for utmost user functionality.

Ideally, we created a distributed operating environment for research and applications to be born, exist, and evolve.

References

[1] Pezely, D.J., *Design Analysis and Potentials*, Human Interface Technology Laboratory, Washington Technology Center, University of Washington, Seattle, WA, 1991.

[2] Pezely, D.J., *Building Multiple, Distributed, Shared, Virtual Environments with Entity Projection and Transportation Facilities*, Human Interface Technology Laboratory, Washington Technology Center, University of Washington, Seattle, WA, 1991.

[3] Leffler, S.J., McKusick, M.K., Karels, M.J., Quaterman, J.S. *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley Publishing Company, New York, NY, 1989, pp. 13, 203-208, 281-310.

[4] Carriero, N., Gelernter, D., "Applications Experience with Linda," *Symposium on Principles and Practice of Parallel Programming, Proceedings of the ACM/SIGPLAN*, Volume 23, Issue 9, September 1988, pp. 173-187.

[5] Forin, A., Barrera, J., Young, M., Rashid, R. "Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach," *Proceedings of the Winter USENIX Conference*, January 1989.

[6] Pike, R., Presotto, D., Thompson, K., Trickey, H., "Plan 9 from Bell Labs," Expected to be published in *1990 UKUUG Conference*, 1990.

[7] 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, 46-63.

[8] Tanenbaum, A.S., *Computer Networks*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[9] Comer, D., *Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[10] Comer, D., *Internetworking with TCP/IP Volume II: Design, Implementation, and Internals*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[11] Spragins, J.D., *Telecommunications: Protocols and Design*, Addison-Wesley Publishing Company, New York, NY, 1991, pp. 118-149.

[12] Pezely, D.J., *Overview of Entities in MOSES*, Human Interface Technology Laboratory, Washington Technology Center, University of Washington, Seattle, WA, 1991.

[13] Pezely, D.J., *The DataSpace Entity Specifications*, Human Interface Technology Laboratory, Washington Technology Center, University of Washington, Seattle, WA, 1991.

[14] Allen, J., *Anatomy of LISP*, McGraw-Hill Book Company, New York, NY, 1978.

[15] Ho, W.W., Olsson, R.A. "An Approach to Genuine Dynamic Linking," *SOFTWARE–Practice and Experience*, Volume 21, Number 4, April 1991, pp. 375-390.