The Persistent Relevance of the Local Operating System to Global Applications

Jay Lepreau Bryan Ford Mike Hibler

Department of Computer Science
University of Utah
Salt Lake City, UT 84112
http://www.cs.utah.edu/projects/flux/
{lepreau,baford,mike}@cs.utah.edu

Abstract

The growth and popularity of loosely-coupled distributed systems such as the World Wide Web and the touting of Java-based systems as the solution to the issues of software maintenance, flexibility, and security are changing the research emphasis away from traditional single node operating system issues. Apparently, the view is that traditional OS issues are either solved problems or minor problems. By contrast, we believe that building such vast distributed systems upon the fragile infrastructure provided by today's operating systems is analogous to building castles on sand. In this paper we outline the supporting arguments for these views and describe an OS design that supports secure encapsulation of the foreign processes that will be increasingly prevalent in tomorrow's distributed systems.[1]

1 Why Global Applications Require More from Local Systems

1.1 The Global Vision

The Call for Papers for this conference envisions that an "as-yet-unbuilt system will allow hundreds of millions of ordinary citizens to access global information and participate in applications of unprecedented scale." Furthermore, it is claimed that the systems software emphasis will shift to providing support for such applications. We agree, but argue that the burden on the software controlling the local machine will only increase, both quantitatively and in some new qualitative ways. Advances in local operating systems are an essential component of future global systems.

1.2 The Hot Distributed Problems

The hot problems raised by this vision of worldwide distribution are a catalog of traditional issues in distributed systems: fault tolerance, mobility, naming, resource location, distributed soft real time, large scale caching and replication, high-speed networking, distributed security, partially federated domains, and varying trust across administrative domains.

All of these problems are certainly real, important, and difficult to solve. However, Mosaic and the Web succeeded spectacularly without addressing a single one of them. In fact, they succeeded *because* they did not address these issues: if they had tried to address them, the software would not have been either portable or easy to install, which were two crucial reasons they spread like wildfire. It has become a cliche to state that the Web is simply a distributed filesystem, but with everything done wrong.

[1] This research was supported in part by the Advanced Research Projects Agency under grant number DABT63–94–C–0058.

Since the Web currently does everything that a distributed system should not, its performance and functionality are limited by those "distributed systems" problems, not local operating systems problems. Transfers are slow for many reasons: connections are not re-used, there is only ad hoc caching and replication, and the link level resources are simply overloaded; multimedia output has no real-time control; the fundamental names are excruciatingly location-based; authorization and authentication are just starting to be deployed; and the lack of fault tolerance is tolerable only because transactions are almost entirely read-only, the standard mode of interaction is interactive, and the human user is always ready to interrupt when problems occur. Resource locating services (Web crawlers and search engines) work well only because the accessible quantity of data is still very small compared to what it will be in five years.

Many of these failings are susceptible to huge improvements with modest effort. For example, caching `http` connections is easy to implement and reaps large benefits [18], predictive caching based on historical access patterns [20] also does very well, per-site caching can be used either with simple proxy servers or with more sophisticated mechanisms such as hierarchical caching [3], and regional mirrors, together with an intelligently ordered resolution of name to address, can be deployed. Link bandwidth will increase dramatically due to skyrocketing demand, increased supply due to recent telecommunications deregulation (at least in the Unites States), and lower costs due to widespread adoption of optical amplifiers on long-distance fiber links.

However, *fixing these fundamental distributed system performance problems will cause the underlying problems of the local operating system to again become critical.* This will be especially true on servers, and indeed, it is already apparent on them [19]. The pressure on both server and client I/O capabilities will be increasingly severe as analog data "goes digital" and demand for bandwidth increases.

On the client side, the vision of the "Internet Appliance" and "Network Computer" is just beginning to be played out. These small, cheap, possibly mobile, stripped down machines will put a premium on efficient exploitation of scarce local resources, including electrical power. Some "thin clients" such as AT&T's combined internet appliance and cellular phone with only 60K of memory [17], will be virtually anorexic. Simultaneously, such thin clients put an increasing burden on servers, as computing, storage and control migrates towards the central servers.

1.3 The Persistent Local Problems

All of these needs of distributed systems make heavy demands on the software local to a node. The heavy I/O requirements may require entirely new approaches to system structure, emphasizing communication over computation, as in the Scout [21] system. Heavy use of multimedia content will require end-to-end Quality-of-Service and resource management. It will do little good to reserve bandwidth in the ATM switch without reserving or scheduling access to the disk, network interface, memory bus, or X display server [14]. Object-oriented middleware like CORBA induces terrible performance problems due to poor match with application needs [7] and no control over the transport protocol [10, 23]. It will be a severe challenge to efficiently use the scarce resources of "thin clients" while simultaneously providing the robust local resource management and generality required by the Internet.

End-system Security

Perhaps most important, local system security is today a disaster, and will inevitably become worse with increased use of machines as nodes in a distributed system. End-system security is obviously in a crisis, with acute security alerts from CERT issued at least weekly. The situation on the most popular OS platforms that don't offer fully protected operating systems, such as Windows or the MacOS, is even worse. Meanwhile, debate goes on about, e.g., browsers' use of SSL vs. S-HTTP. We believe that such decisions matter little when end-system security is abysmal. For example, even relatively robust authentication mechanisms such as Kerberos store their session keys in locations vulnerable to local intruders [2], e.g., a local file or in shared memory. Even attacks from the local LAN may threaten these keys; for example, local memory may be paged out from a diskless workstation or to network RAM and thus be vulnerable to network sniffing.

Today, local OS security *mechanism* is lacking in numerous areas. Besides the vulnerabilities to confidentiality and integrity that are revealed weekly, local systems are so vulnerable to attacks that can crash them or their critical components, i.e., attacks on availability, that such problems don't even warrant announcements from CERT. Finally, local systems make almost no attempt to address the problems of denial of service or covert channels. In other words, true isolation between subsystems cannot be achieved by today's systems.

However, security *policy* is in even worse condition. This has been evident to the security research community for years. For example, on ordinary non-distributed systems it is apparent that we have no good handle on composing and managing policy, but instead merely have a mishmash of configuration files and access modes. Out of this mishmash of interacting state, many security breaches arise. So far, the relevant formal security work on composability has been too abstract to offer much hope.

The bankruptcy of policy has recently become obvious to a wider community, due to issues surrounding "executable content." In Java [12] or SFI-based [1] schemes for safely executing untrusted foreign code, there is currently no effective way to resolve the tension between functionality—doing anything useful—and protection. These systems are useful only because, for the most part, they simply animate data on the client's screen. When they need to do something more interesting they typically query the user for an exception to the blanket policy, e.g., against reading local files. It is a very long way from this moderately useful but tedious and limited policy, to the complex policy required to fulfill a vision of applications cooperating across the world, engaging in complex compute tasks, without a human in the loop on every file open call.

Java

The Java language does offer type-based memory and interface protection, but attempts to solve only part of the security problem. In particular, its runtime environment [26] provides little way to control memory or cpu consumption, aside from ordinary pre-emption. Even for what it's supposed to control, the fundamental design of the Java security mechanism has many deficiencies, as detailed by Dean et al [5]. Many of these basic design deficiencies are the underlying causes of holes that have already been exploited. These deficiencies include such fundamental problems as: (i) the Java SecurityManager is supposed to be a *reference monitor*, but is not: it is not always invoked, it is not tamperproof, and it is not susceptible to complete analysis and test; (ii) there is no identified *trusted computing base*, but instead, "substantial and diverse parts . . . must cooperate to maintain security." These problems and others make it unlikely that Java-based systems will ever provide truly robust security. Since Java-based security is obtained in an ad hoc manner, similar to "security" on today's Unix systems, it is likely that it will show similarly fragile behavior. This is especially likely to become true when more aggressive security policies are used, and when the complexity of the environment inevitably increases, due to larger numbers and sizes of interacting Java-based components.

One strength that the Java virtual machine does provide today is portability of object code. This has obvious advantages for global applications. However, before Unix splintered, Unix on Vaxen once provided widespread portability of object code, as does object code for Windows today. It is not clear that similar splintering will not happen to the JavaVM, or that the number of machines running, say, Win32/x86, will not effectively provide similar portability of ordinary machine code.

Related to its portability advantage, an obvious limitation of Java-based security is that it is limited to programs written in Java (actually, it is limited to languages compilable to the Java virtual machine). In today's networked environment, users frequently run ordinary binaries off the net, or compile from unchecked source, and hope that the programs are harmless. A secure, isolated, controllable environment, that can run an arbitrary subsystem with full speed in the common case, doesn't require the target program to be in a special language, and is based on a design that makes robustness likely, would be of great use in future distributed systems.

2 A Local Operating System for Distributed Systems

2.1 Recursive Virtual Machines

In the sections above we have articulated some of the local needs of machines that result from their participation in future distributed systems. Partly in response to these needs, especially in the security realm, we are developing a new operating system that can efficiently support a *recursive virtual machine* (RVM) execution model. Most modern operating systems provide a concept of virtual machines—e.g., processes or tasks—and allow several such virtual machines to coexist on a single machine and compete with each other for hardware resources. However, our OS architecture is unique in that it allows virtual machines to completely *contain* other virtual machines—i.e., it supports recursion.

To illustrate this concept, suppose one were to take a PC running, say, Linux, and in a process on that Linux box one were to run an x86/PC hardware simulator. Suppose the PC hardware simulator is complete enough that it can run a real operating system. So one boots up a second copy of Linux running in this machine simulator, and then when *that* copy of Linux comes up, one runs *another* machine simulator in that one, running another copy of Linux. . . .

Such an arrangement of machine simulators would have little practical value because performance would be dismal even one level deep, and would become exponentially worse as more layers were added. However, if the performance problem didn't exist, the arrangement has a number of useful properties:

- Parent virtual machines can control *every aspect* of the execution of their child virtual machines: all processes, threads, etc. are fully visible to and modifiable by the parent.
- The child can only access or consume resources that the parent itself already owns. For example, any memory that the child can access is simply part of the data area of the parent's machine simulator. Similarly, any CPU time that the child uses was "donated" by the parent by the virtue of the parent running the machine simulator code.
- The child virtual machine, and any further descendents it may contain, are completely encapsulated within the parent's virtual machine, and are invisible to entities outside the parent's virtual machine (or, at most, visible only as "ordinary dumb data" in the parent's virtual address space). For example, if one runs a '`ps`' command in the top-level Linux environment, it will show only a single process for the next-level virtual machine. This "encapsulation property" provides a number of simplifying benefits, which we discuss later, in Section 2.3.

The basic goal of our OS architecture is to provide an environment with all of the important properties such a nested machine-simulator arrangement would have, *without* instruction set emulation and the corresponding loss of performance. Of course, a few technical restrictions must be made—e.g., a child virtual machine must use the same processor architecture as its parent—but most of the important practical properties of the model are still achievable.

Note that the concept of recursive virtual machines has analogies to Unix's hierarchical process organization, in that parent processes can create and control child processes. However, the Unix model falls far short of a true RVM model, in at least the following important respects: (i) Parent processes have only a very limited degree of control over their children. (ii) Child processes can allocate and use resources that the parent process doesn't own (and possibly never did). (iii) The child will persist after the parent exits. (iv) All processes are globally visible in a single process ID namespace. This doesn't mean that the Unix process model isn't useful—in fact, it is very useful. However, a true RVM model makes it possible for an arbitrary user process to completely control its descendants. This provides the flexibility and power that allows a process at any level of the system to isolate and control arbitrary subsystems: a facility that is needed to securely run arbitrary untrusted code.

2.2 A Software Virtual Machine Architecture

In the 1970's special *virtualizable hardware architectures* [11, 15] were proposed, whose goal was to allow software virtual machines to be stacked much more efficiently than on normal hardware. Our approach is to design a virtualizable architecture with the same goal of efficient layering, but one that is appropriate for *software* implementation. The three components of our virtualizable architecture are the standard non-privileged machine instructions, the operations exported by our Fluke microkernel [6], and a set of higher-level "Common Protocols." Virtual machine monitors (VMMs) executed on this virtual machine can efficiently create additional, recursive virtual machines in which applications or other VMMs can run [8].

Kernel Properties

The Fluke microkernel provides simple memory management, scheduling, and IPC primitives similar to those of conventional "small" microkernels such as the V++ CacheKernel [4], L3/L4 [16], and KeyKOS [13]. The Fluke kernel API does not *enforce* the RVM model, but it *enables* the model—the ability for any process to completely control its children— by providing three vital properties:

- All kernel primitives are completely *relative*, implying no global resources, namespaces, or privileges. This is required so that each process sees exactly the same Fluke API and execution environment, and can function as a fully privileged entity over its descendents.
- All primitive kernel objects (e.g., threads, mappings) are *owned* by, or associated with, specific processes. This is required so that a process can *find* all of the resources used by its children. Our Fluke implementation obtains this property by associating kernel objects with small chunks of normal user memory. Fluke provides a kernel primitive to return pointers to all of the kernel objects associated with a particular range of user memory; since a child's memory is by definition contained within its parent, this makes it easy for the parent to locate all of the kernel objects. To further control its children by constraining object creation, a parent can remove the special "object_create" virtual memory permission attribute on portions of its children's memory.
- All state contained in primitive kernel objects is *exportable* as plain data, in a form that ordinary programs can later use to regenerate the objects. This is required so that any process can both get and set the complete state of its children, once it has located their primitive objects.

The microkernel's API supports efficient recursion (hierarchical process structuring) in several ways. For memory resources, the virtual machine hierarchy gets explicit support from *relative* memory mapping primitives that allow address spaces to be efficiently composed from other address spaces. For CPU resources, the kernel provides a primitive that supports hierarchical scheduling models. Such schedulers are easily implementable by ordinary user processes. To allow safe short-circuiting of the hierarchy, the kernel provides a global capability model that supports *selective* interposition on communication channels.

Higher Level Common Protocols

The capability model is exploited by the "Common Protocols," a set of well-defined IPC interfaces that provide I/O and resource management functionality at a higher level than in traditional virtual machines, more suited to the needs of modern applications: e.g., file handles instead of device I/O registers. The Common Protocols define how children get higher-level resources from their ancestors. A process's "parent port" is the highest level interface used for parent/child communication, effectively acting as a "name service" through which the child requests access to all other services. This is the only interface that *all* VMMs interpose on; a VMM selectively interposes on other interfaces only as necessary to perform its function. The overhead of this interposition is minimal, because typically only a few requests are made on the parent interface during the child's initialization phase, to find other interfaces of interest. The parent interface currently provides methods to obtain initial file descriptors (e.g., `stdin`, `stdout`, `stderr`); find a filesystem manager, find a memory manager, find a process manager, and to exit.

During the service discovery ("binding") phase, an important feature of this model happens automatically: "cut through" of irrelevant middle layers. If a process is functioning as a VMM, it receives initial requests for service classes on its child port. If that VMM is not modifying the behavior of that service class—that resource (e.g., memory, or files, or process management)—it simply returns the port that *it* is using to obtain that service, which it of course obtained from its own parent at initialization time. From then on, requests from the child for that service don't involve the parent in any way, but go straight to wherever that parent's particular resource port points, which may be the grandparent or may be further up the tree. If the parent *is* modifying the resource, e.g., it's a virtual memory manager that is turning physical memory into virtual memory for its children, then it passes down a reference to a new port of its own, and services memory requests itself. In this way we obtain automatic "cut through" of the hierarchy of virtual machines—a key factor in the efficiency of this model. Thus, a complete virtual machine interface is maintained at each level, and efficiency derives from needing to implement only *new* or *changed* functionality at a particular level.

2.3 Security through Recursive Virtual Machines

In this section we outline some of the security-relevant features of our RVM model, focusing on mechanism, not policy. The RVM model (i) is flexible: can be applied *by* any process *to* arbitrary sub-environments, (ii) is efficient: interaction *inside* an environment need never involve the security manager, and (iii) can provide strong resource accounting and control. We elaborate on these features below.

Our proposed use of virtual machines for security is well established: one of the uses of "classic" virtual machines was to provide isolation guarantees between subsystems [22]. However, we also provide the ability to *nest* virtual machines, and that is important for "worldwide applications." A machine that runs untrusted applications requires a number of features from its operating system. It needs the ability to, by default, isolate the untrusted environment from the rest of the machine, and control any interaction with the rest of the system that it decides to allow. It needs the ability to control resource use by the foreign process, including CPU and memory. Finally, it needs to be able to provide these same facilities *flexibly*: to arbitrary user processes, such as browsers, that need to control their own children. Furthermore, in the future, there will exist distributed systems consisting of multiple layers of loosely coupled interacting objects. Thus a foreign applet might invoke, from a third site, an applet foreign to *it*. When the first applet does so, it will itself need the ability to control the second applet's resource use, and so on. The RVM model is obviously well matched for such arbitrary nesting of security and management domains.

Thus an ordinary user can create protected sub-environments in which arbitrary "untrusted" programs can be run without giving them access to all of the user's files and privileges. The RVM model takes the well-known concept of a "separation kernel" whose function is to separate information domains, and generalizes it to provide a flexible number of separation kernels on the same machine.

The "encapsulation property" (that the state inside a virtual machine is invisible to outside parties) has two primary benefits. First, it vastly simplifies the management of arbitrary multi-process environments. Handling dynamic multi-process environments in normal OS's like Unix is always a major complication, and many monitoring schemes simply can't do it. In the RVM model, this whole problem never arises. Second, operations among the entities inside a particular virtual machine go at full speed. They can communicate freely among themselves, without the parent ever getting involved Since all communication into and out of an environment can be monitored without having to keep track of everything that happens *within* that environment, entire subsystems running on a machine can be isolated from each other cleanly, with only well-defined communication allowed between those subsystems. For example, a "same-machine firewall" between less trusted and more trusted applications could easily be implemented.

The RVM model can cleanly provide strong resource accounting and control. Most traditional kernels do not deal well with the issue of how to account for the system resources (especially memory) used by all of the different OS and user processes present in the system, since each process is generally able to allocate resources largely independently of each other. In a system such as Unix or Mach, although one process can fairly easily monitor the activity of a child process it directly creates, it is extremely difficult to monitor the activity and resource use of "grandchild" processes created by that child process. Similarly, if the parent process kills the child process, it still can't be sure all activity started by that child has been terminated. Under the RVM model, this accounting and control can be provided in a relatively straightforward manner.

Outside the security realm, but relevant to wide-area applications, is another feature of such isolation and resource control. That control is also useful for resource reservation, as when guaranteeing a certain amount of physical memory to real-time applications. Applications with soft real-time constraints will likely play a dominant role in the future.

The RVM model can also address covert channels. Storage channels should not be a problem, because the parent has complete control over what the child sees: any collaboration can be throttled. Timing channels are controllable as well, since arbitrary scheduling policies can be provided by user-level threads [9]. If the bandwidth of covert timing channels needs to be minimized. these policies can be hierarchical, enforcing the RVM model, and can use arbitrary algorithms, such as forcing fixed-slice scheduling, where the scheduler soaks up any unused cycles in a slice before switching to a thread in another security context.

2.4 Related Work

Our RVM model is similar to what the Cambridge CAP Computer [25] provided, although inefficiently, with the aid of hardware supported "indirect capabilities." The CAP Computer supported an arbitrarily-deep process hierarchy, in which parent processes could completely virtualize the memory and CPU usage of their child processes, as well as trap and system call handlers for their children. However, the CAP computer enforced the process hierarchy *strictly*, and did not allow communication paths to "cut through" the layers as our system does. As noted in retrospect by the designers of the system, this weakness made it impractical for performance reasons to use more than two levels of process hierarchy (corresponding roughly to the "supervisor" and "user" modes of other architectures); thus, the uses of recursive virtual machines were never actually explored or tested in this system.

Wagner et al [24] describe a tool that interposes on a child process's Unix system calls, using the Unix process debugging facilities. They have concentrated on providing useful functionality while working within existing operating systems, and apparently achieved this. Their goal of providing the ability to run untrusted binaries is the same as one of our goals. Besides a totally different technical approach, there are many other differences. They interpose on kernel services, which is similar; however, they have no way to universally and reliably interpose on all kernel services, since their interposition is (apparently) driven by *a priori* knowledge of the list of possible system calls. Their system is vulnerable if the underlying operating system were to implement a new system call of which they were unaware. Inevitable version skew is highly likely to induce this situation, and probably already exists with undocumented calls. Our design provides a choice at what level to operate: a mediating task could operate at a low level, enforcing mandatory access control with a flexible policy, or it could operate at the high-level OS personality (e.g., Unix) level, as they do, but by also controlling all IPC operations, ensures that no new OS services get through. They offer much less control over resource use than do we: standard Unix does not provide the ability to change the resource limits on a running process, provides no ability to limit the paging *behavior*, just the total amount of memory (so, for example, a malicious process might cycle through its memory in a way designed to cause thrashing), and does not provide flexible scheduling among children, but only limits on the total amount of cpu time. It is difficult in their system to provide single-point control over a multi-process environment, because the monitoring process must replicate itself when its child forks. A fundamental property of our design is that operations *within* a nested environment are distinguished from operations across environments. By contrast, they have to check every kernel operation. Finally, our overall design is oriented to providing support for highly-efficient interposition on IPC and other kernel services. Although the performance they measured was excellent, we believe this was because their example applications exhibited a low ratio of system calls to actual computing.

2.5 Status

We have defined the detailed Fluke API [6], the "Flexible μ-kernel Environment." A prototype Fluke implementation is running on the x86 platform, along with several virtual machine monitors, including a demand paging virtual memory manager, a checkpointer, a process manager providing a subset of POSIX functionality, and a transparent debugger. Later this year we will be constructing a security manager that provides an encapsulated environment within which to run untrusted applications. We plan to first implement a browser's Java applet "security policy" as an example, and then move on to more complex policies. Collaborators from the U.S. National Computer Security Center are adding support for traditional subject-based security. Our intent is to develop a means to virtualize the ensuing security identifiers, preserving the "relativistic property" of the interface. Finally, we expect to make a formal release of Fluke in this calendar year.

3 Conclusion

We believe that although the burgeoning worldwide network will demand crucial advances in many areas of distributed systems, it will also make increasingly heavy demands on the local operating system. In the area of isolation and resource management, it places new demands. We believe we have developed an operating system architecture that can efficiently address that demand for isolation and resource control.

References

[1] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and Language-Independent Mobile Programs. In *Proc. ACM SIGPLAN Symp. on Programming Language Design and Implementation*, pages 127–136, May 1996.

[2] S. M. Bellovin and M. Merritt. Limitations of the Kerberos Authentication System. In *Proc. of the Winter 1991 USENIX Conference*, pages 253–267, 1991.

[3] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proc. of the USENIX 1996 Technical Conference*, pages 153–163, Jan. 1996.

[4] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 179–193. USENIX Association, Nov. 1994.

[5] D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proc. of the 1996 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1996.

[6] B. Ford and M. Hibler. Fluke: Flexible μ-kernel Environment — Application Programming Interface Reference (draft). 121 pp. University of Utah. Available as ftp://mancos.cs.utah.edu/papers/sa-flukeref.ps.gz and http://www.cs.utah.edu/projects/-flux/fluke/html/sa-flukeref/ (HTML format), 1996.

[7] B. Ford, M. Hibler, and J. Lepreau. Using Annotated Interface Definitions to Optimize RPC. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, page 232, 1995. Poster.

[8] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, S. Goel, and S. Clawson. Microkernels Meet Recursive Virtual Machines. Technical Report UUCS-96-004, University of Utah, May 1996.

[9] B. Ford and S. R. Susarla. Flexible Multi-Policy Scheduling based on CPU Inheritance. Technical Report UUCS-96-005, University of Utah, May 1996.

[10] A. Gokhale and D. C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In *SIGCOMM '96*, San Francisco, CA, August 1996. ACM.

[11] R. P. Goldberg. Architecture of Virtual Machines. In *AFIPS Conf. Proc.*, June 1973.

[12] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems Computer Company, 1996. Available as http://java.sun.com/doc/language_environment/.

[13] N. Hardy. The KeyKos Architecture. *Operating Systems Review*, September 1985.

[14] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera III. Modular Real-Time Resource Management in the Rialto Operating System. In *Proc. Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[15] H. C. Lauer and D. Wyeth. A Recursive Virtual Machine Architecture. In *ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, pages 113–116, March 1973.

[16] J. Liedtke. On Micro-Kernel Construction. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, Dec. 1995.

[17] J. Markoff. AT&T Plans to Offer Internet Over a $500 Wireless Phone. *New York Times*. July 12, 1996.

[18] J. C. Mogul. The Case for Persistent-Connection HTTP. In *Proc. of the SIGCOMM '95 Conference*, pages 299–313, Aug. 1995.

[19] J. C. Mogul. Operating Systems Support for Busy Internet Services. In *Proc. Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[20] J. C. Mogul. Hinted Caching in the Web. In *Proc. of the Seventh ACM SIGOPS European Workshop*, Sept. 1996.

[21] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. Technical Report 96–05, University of Arizona, Dept. of Computer Science, May 1996.

[22] G. J. Popek and C. Kline. Verifiable Secure Operating Systems Software. In *AFIPS Conf. Proc.*, June 1973.

[23] D. L. Schmidt, T. Harrison, and E. Al-Shaer. Object-Oriented Components for High-Speed Networking Programming. In *Proc. of the USENIX Conference Object-Oriented Technologies*, June 1995.

[24] D. Wagner, I. Goldberg, and R. Thomas. A Secure Environment for Untrusted Helper Applications. In *Proc. of the 6th USENIX Unix Security Symposium*, 1996.

[25] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System*. North Holland, NY, 1979.

[26] F. Yellin. Low Level Security in Java. In *Proc. 4th Int'l World Wide Web Conference*, pages 369–379, Dec. 1995.