Operating Systems for Component Software Environments

Noah Mendelsohn
Lotus Development Corporation
One Rogers Street
Cambridge, MA 02142

Abstract

Although component software has emerged as one of the most significant and commercially successful technologies of the past few years, few operating systems are designed to host and manage component software effectively. Components impact OS architectures in areas of security, process isolation, code sharing, installation management, and user interface design. A more radical question is: can effective operating systems be built of modular, interchangeable component parts? The thesis of this paper is that effective support of components is a key requirement for operating systems of the future.

1. Introduction

Component software is revolutionizing the application industry: from word processors to back end servers to web browsers, a broad range of commercial applications is being assembled from ActiveX's [3, 7], OpenDoc parts [15], Java Applets [4], Java Beans [13] and OCX's. There are rumored to be over a thousand businesses developing reusable commercial components, and special purpose controls are emerging as the new building blocks for software reuse in vertical markets. Microsoft has used its ActiveX infrastructure to muscle into the Web browser business, with an offering that can be broken into modular pieces useable on both the OS desktop and in special purpose applications [6, 8]. Java Applets and Java Beans are emerging as the first breed of truly portable components, which can run across a range of operating systems and devices.

Ironically, all of this is happening just as a generation of operating systems designed to support more conventional software is maturing. Much of OS research remains focused on issues arising from these earlier systems: improving the performance of networked file systems, exploring impacts of scheduling policies on processor cache miss rates, etc. Though most of these remain important problems in a world of components, the thesis of this paper is that components introduce a range of new issues and architectural requirements, and that they change the environment in which other questions, such as performance, should be explored. Finally, if applications can successfully be assembled from modular components built by separate companies and organizations, can operating systems be built in the same manner?

2. A brief history component software and operating systems

Many techniques have evolved for building code that is modular or reusable. Using such techniques, one can build software *components* [5], which are designed and packaged for reuse, typically by multiple independent organizations. Reusable software components are often compared to their mechanical counterparts, which in the 19th century were a catalyst for the industrial revolution. Indeed, many of the operating system issues discussed in this paper have direct counterparts in the mechanical world: tracking inventories of mechanical components versus managing the distribution and registration of software components; keeping documentation and run time type information in sync with the corresponding component or mechanical design; qualifying a component for use in specific environments (consider, for example, an insurance application program which can be built only of components which have been certified by the actuarial department.)

The earliest software components were subroutines, typically shared in source form by duplicating card decks or in other ad-hoc ways. Almost immediately, operating systems adapted to facilitate the use and sharing of such software. File systems stored the code, and link editors allowed application programs to be assembled from shared pieces, first in batch mode, and later through dynamic

linking at execution time. Operating systems soon provided explicit services for loading libraries, managing dynamic links, and for sharing in-memory library images across multiple applications and processes. For many years, subroutines and code libraries were the fundamental building blocks for shared code. Distribution of code libraries was still ad hoc, often on magnetic tape, with installation and management procedures particular to the library: a math library might be installed and registered in one manner, but a spelling checker would be installed specifically into a word processor.

Unix standard I/O and pipes [11] encouraged the use of entire applications as components. Utility applications such as *sort* could be assembled to make larger applications for special purposes. Indeed, certain applications such as *tee* were intended for use only as part of a larger "pipe". Once again, a combination of language and operating system technologies were used to implement the modularity required of components. Per component overhead and limitations of the stream abstraction prevented pipes from evolving into a more general purpose component framework.

Device drivers were and are another special purpose example of software componentry in traditional operating systems. Drivers are typically packaged and installed in a standard manner, have a tight contract with the OS, and are in some sense interchangeable.

Except for subroutines and code libraries, none of these technologies provided a general or ubiquitous framework for the development of modular software and shared code.

3. Modern component software

A new style of component has emerged to support tight integration of fine grain components from diverse suppliers. Examples include Microsoft's OLE/OCX/ActiveX [2, 3, 7], OpenDoc from CILabs [15], and more recently Java Applets and especially Java Beans from Sun Microsystems [4, 13]. Characteristics typically shared by these component architectures include:

- Use of object oriented API's to provide (1) a richer "contract" for service (compared with simple subroutines, pipes, etc.) ... typical features include component *properties*, *methods*, *events* and some standard means of reflecting errors; (2) a means by which components can be extended or specialized to create new components; and (3) a means by which multiple components can be assembled to create richer components.
- A binary calling standard for mapping to each particular platform. If various components and languages implement the calling convention, then all operate interchangeably. Microsoft, for example, supports full interoperation of components written in C, C++, Java, Basic, and Cobol on 32 bit Windows platforms. Java Beans are unique in supporting a common binary standard for all platforms – a single packaged component can therefore be used in any application on any platform, as well as in cross platform applications.
- Support for recursive activation of one component within another.
- Standard persistence mechanisms.
- Machine readable meta-class descriptions (run time type information) to support scripting, builder tools, and strongly typed dynamic assembly.
- A name space in which component classes are identified. Typically, the name space must support dynamic creation of globally unique component names by organizations dispersed throughout the world. Examples include OLE Class Identifier (CLSID) Globally Unique Identifiers (GUIDS) [2], which are based on DCE UUIDs; and Java class names, which are based on DNS names and are used as the name space for Java Beans. Components are activated by name from these name spaces, not by file identifier.
- Standard packaging and/or registration technology, typically including: (1) a means by which all the files and data required by the component can be packaged, distributed, installed, and identified in a standard manner, regardless of the purpose of the component; and (2) licensing and other information needed to determine whether a component can legally be used, whether it has been qualified or approved for a specific purpose, etc.
- A common means of activating or instantiating components, regardless of type or intended use.

These technologies are being successfully applied to a broad range of server and client components for a wide variety of applications.

4. Challenges for operating system architecture

The new component architectures impact operating system usage in many ways. Code is loaded not by file name, but by component name. Resources must be allocated to and shared by the multiple components comprising an application. Security, process isolation, code sharing, code segment and code image file management, code installation, and user interface design must all be considered.

A look at Microsoft's ActiveX architecture highlights some of the issues: in practice, ActiveX components are packaged in Windows dynamic link libraries (.dll), which are loaded and managed by the OS in the same manner as any other library. The multiple components that form a single application share a process context, an OS security context, execute on the same stack(s) and share access to the same memory.

The OS is oblivious to the component abstraction and cannot effectively provide a service tailored to individual components. In a component-based application, a file opened by one component might inadvertently be manipulated or closed by another. Use of a shared library .DLL by one component can prevent correct execution of a second component that depends on a revised version of the same library; library loading is a service provided to processes not, individual components. On the Windows 3.1 platform, arbitrary nestings of OCX components from multiple vendors share a stack of limited size; there is no way to know in advance how much, if any, memory is available for any particular component. Building a robust component therefore becomes extraordinarily difficult. Further, lack of isolation tends to result in one component's bugs crashing another: component 1 almost fills the stack, the crash comes when the user activates component 2. In a commercial setting, the support costs of dealing with such confusion are significant. These examples, based on OCX and ActiveX, illustrate the manner in which a component's "contract" with the outside world extends far beyond its explicit API; component vendors must know when and where their components will run safely. In practice, Microsoft has added certain component management functions to the Windows NT and Windows 95 kernels, but many of the problems discussed here remain. Similar difficulties are observed when other component architectures, such as OpenDoc, are hosted on conventional operating systems.

All of these problems result from a poorly specified contract between a component and the supporting OS, compounded by the need to integrate components from multiple suppliers into a single application. The challenges for component vendors are significant. How can one effectively test and certify the reliability of a component that will run in such an unstructured environment? How can potentially sensitive data be safeguarded from other components in the same application?

It's quite an irony that, just as operating systems are incorporating increasingly robust isolation across processes and users (C2 security, ACL's, etc.), modern component architectures are collapsing the execution environment into fewer processes. Most of the code executed by a modern Web browser, including plugins (another form of component), ActiveX controls, etc, often for multiple end-user applications, runs in a single OS process and protection domain. Indeed, the Java sandbox model, which does deal with some of these problems, underscores the inability of traditional OS's to provide an effective solution.

Just as inventory management is a central problem when manufacturing mechanical assemblies, component management is an increasingly important challenge for the modern networked OS. Application builder programs (PowerBuilder, JBuilder, Visual Age, Visual Cafe, Visual Basic, etc.) depend on libraries of components which can be assembled to form applications. These component *repositories* are evolving into secure, distributed, code and data stores which manage the installation and deployment of components throughout a network. The new repositories assist with internationalization of components, license tracking, certification management etc. Is an "a.out", ".exe", or ".dll" file still the right core OS abstraction for loading code, or should there be a more fundamental notion of *component* which can be bound and activated? Is the file system the right code storage model, or is it to be replaced by a distributed database or registry, tuned to the management and deployment of these components?

Dynamic linking, at the OS level, is usually procedural, yet components are increasingly object oriented. Will new OS's directly support object composition and dynamic method dispatch, or are the existing layerings of object "sugar" onto procedural plumbing still appropriate? As operating systems evolve to provide separate execution domains for untrusted components, how will performance be maintained? Can debugging be done effectively in such an environment?

5. Operating systems for component applications

The section above presented a sampling of the many ways in which component software differs from that

originally supported by operating systems such as Unix and NT. One may reasonably guess that, as components become ubiquitous, *component* will join *file* and *process* as a fundamental OS abstraction. In the same sense that a process represents a clearly defined execution environment for traditional software, components will have a useful and well specified contract with the host operating system, as well as with other components in the same application. Specifics of the implementation will depend on the nature of the system and its intended application. Even today, a real-time OS provides a higher performance but less robust process abstraction than one would find in Unix or NT. Similar considerations will apply when tailoring the component abstraction to particular requirements.

The tradeoff between component isolation and performance will also depend on the types of components and applications to be supported. Indeed, there may be value in having two levels of component within the same OS, much as one has both threads and processes. Families of mutually trusting lightweight components, operating with minimal isolation and highest performance, would be aggregated to build course grained mutually-suspicious components, from which applications would ultimately be assembled.

Over time, dynamic link libraries will be replaced entirely by components. Each will be named by a component identifier, an URL perhaps, and retrieved from a distributed component repository. Microsoft's Component Object Model, which underlies ActiveX, already uses the Windows NT registry as a primitive implementation of a component repository. Most likely, object oriented dynamic binding and aggregation will be integrated into the core OS services for loading libraries and components.

The Java Virtual Machine [10, 14], viewed as an OS, is an interesting first step down the path outlined in this paper. All Java code is object oriented and is bound and activated by class name; dynamic object based linking is provided as a core OS service. Although Java provides strongly specified notions of *class* and *object*, libraries (Zip and Jar files [12]) and components (Java Beans [13]), are treated informally, or layered on top of the core OS abstractions. Isolation is supported by abstractions of ClassLoader and ThreadGroup, which are only indirectly related to the concept of component or Bean. Beans are packaged in Jar files, which are analagous to Unix object code archives or Windows object code libraries, but facilities for managing or binding to packaged Beans are minimal. There is no clear notion of a component repository, except insofar as ClassLoaders provide an enabling building block. Many other issues relating to component management remain unresolved; as far as the Java VM is concerned, a Java Bean is just another object.

6. Conclusion

The challenges of building component based applications are increasingly well understood, but the required evolution of the OS layer has just begun. This paper outlines some of the issues that must be considered, but detailed solutions will come from others in the future.

A formal abstraction for *component*, joining those traditionally provided for processes, threads, files, etc., seems to offer a framework within which component software can effectively be supported.

The great success of Java itself, and the initial positive response to Java Beans, suggests that a viable operating system can be based on the concepts outlined in this paper. Certainly, offerings such as OCX, ActiveX, OpenDoc and Java Beans have proven the importance and commercial viability of component software, and Microsoft's Windows NT implementation has demonstrated both the potential and the limitations of a component system layered on a traditional operating system. Indeed, based on experience with NT, it is premature to conclude that modern operating systems must be replaced, or even radically altered to support components. Perhaps a gradual evolution will be more appropriate. It is not premature to make components an important new focus for operating research and development. Whatever the final answer, the thesis of this paper is that the "component revolution" will be a defining backdrop for OS evolution and research during the next few years.

7. Epilogue: the component OS

If application components have proven the possibility of constructing large software projects out of modular pieces, some of which are "off the shelf" components, could a radically new OS be built in the same manner? What if more than device drivers, shells and file systems were pluggable? Would the packaging and programming techniques used for application components apply? Could a customized OS be developed using visual tools and with minimal programming skills? If we could do it, would we be solving a useful problem?

Obviously, some of this is already happening. The user interface for modern commercial operating systems is increasingly modular and replaceable: Apple and IBM used OpenDoc in Copland and the OS/2 Workplace shell respectively, and Microsoft's next Windows shell will make heavy use of replaceable ActiveX components. These components can already be assembled with powerful visual tools, involving little or no programming.

But what about core functions? SPIN [1], Exokernel [9] and other extensible kernels are exploring the infrastructure required for safe, modular extension of a protected operating system. They are laying the groundwork for implementation of core operating systems features using components. So far these systems have apparently not explored in detail issues of packaging, distribution, visual assembly, or integration with application level component models.

Will the buffer manager ever be an off the shelf component in the file system? Will thread management come in a box? Returning to analogies from the industrial world, automobiles are built from many thousands of components. In the warehouse and on the assembly line these are tracked through a uniform component naming scheme by integrated tracking systems. Certain characteristics, such as weight, are recorded and processed in a common manner, regardless of component type or supplier. Yet, many of the components in a car are special purpose, used in only one type of car and tightly dependent on the components around them. Others, such as standard nuts and bolts, are interchangeable with other cars, and even with other manufactured goods. In the middle ground are items such as steering wheels, which often have limited applicability across a group of vehicles with similar requirements. Some, such as radios, can be replaced with aftermarket equivalents after the product has been built and sold.

The analogy to operating systems may be a good one. Over time, the general techniques of standardized packaging, identification, and certification can be applied uniformly to OS and application components. Component activation will be a core OS service, for the kernel as well as users, and where possible pieces of the OS will be named, managed and activated as components. As with cars, many pieces will be specific to individual OS's, tightly integrated for performance and robustness with the components around them. Over time, some new pieces of the OS will become interchangeable, and available "off the shelf", but many others will remain special purpose.

Whether the complexity of such an OS implementation is justified remains to be seen. Yet, it is interesting to speculate that in the future, operating systems may be assembled or configured graphically, and with relatively little programming, and that the benefits of interchangeable commodity parts may eventually be applied to all layers of the software hierarchy.

8. Acknowledgments

Mark Day and the conference referees have provided many helpful comments on the manuscript of this paper. For the past four years I have had the pleasure of working with and learning from some of the pioneers in the component software industry. Though our contacts are occasional, Tony Williams, Kurt Piersol, Graham Hamilton, Larry Cable and other members of the OLE, OpenDoc and Java Beans teams have taught me, directly and through their work, much of what I know of this field. Douglass Wilson, Barry Briggs, Joe Guthridge, Jack Ozzie, and a long list of co-workers at Lotus have helped me to learn the realities of building world class applications, components and objects, and supporting them for millions of customers. Special thanks to Dean Burson, to all the members of my team past and present, and especially to Alex Morrow.

9. References

[1] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiucznski, M. E., Becker, D., Chambers, C., and Eggers, S., *Extensibility, Safety and Performance in the SPIN Operating System*, in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pp. 267-286, December, 1995.

[2] Brockschmidt, K. *Inside OLE 2*, Microsoft Press, 1994.

[3] Chappell, D. *Understanding ActiveX and OLE*, Microsoft Press, 1996,

[4] Cornell, Gary, and Horstmann, Cay S., *Core Java*, SunSoft Press, 1996.

[5] Cox, Brad J., and Novobilski, Andrew J., *Object Oriented Programming: An Evolutionary Approach*, Addison Wesley, 1991.

[6] DeLascia, P. and Stone, V. *Sweeper*, Microsoft Interactive Developer, Vol. 1, No. 1, pp. 16-52, Spring, 1996

[7] Denning, A., *ActiveX Controls Inside Out*, Microsoft Press, 1997.

[8] Eddon, G. and Eddon, H. *Preview Version 3.0: not just another Web browser*, Microsoft Interactive Developer, Vol. 1, No. 2, pp. 14-18, Summer, 1996.

[9] Engler, D., Kaashoek, M.F., and O'Toole, J., *Exokernel: An Operating System Architecture for Application-Level Resource Management*, in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pp. 251-266, December, 1995.

[10] Gosling, J., Joy, B. and Steele, G. *The Java™ Language Specification*, Addison-Wesley , 1997.

[11] Kernighan, B. and Pike, R., *The UNIX Programming Environment*, Prentice-Hall, 1984.

[12] JavaSoft. *The Jar Guide*, Unpublished. Available from: http://www.javasoft.com/products/jdk/1.1/docs/guide/jar/jarGuide.html, December, 1996,

[13] JavaSoft. *The Java Beans™ 1.0 API Specification, Version 1.00-A*, Unpublished. Available from: http://java.sun.com/, December, 1996,

[14] Lindholm, T. and Yellin, F., *The Java™ Virtual Machine Specification*, Addison-Wesley, 1997.

[15] Orfali, R., Harkey, D. and Edwards, J., *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, 1996.