Building Appliances out of Components using Pebble

Kostas Magoutis†*, José Carlos Brustoloni, Eran Gabber, Wee Teck Ng and Avi Silberschatz

†Division of Engineering and Applied Sciences
Harvard University

Information Sciences Research Center
Lucent Technologies - Bell Laboratories

1. Motivation

Appliances are special purpose systems that offer high processing speed, ease of configuration, safety, fault isolation, and minimal need for administration by human experts. Traditional approaches to building an appliance operating system have been either building it from scratch [CacheOS] or stripping down a monolithic kernel to its basic components [Jaeger99]. The former approach is costly and the resulting product is likely to be highly specialized and not easily extensible. The latter approach is not easy, as the OS code and data structures are often shared and closely intertwined. The resulting OS is also likely to be coarse-grained and not easily customizable. Most appliances are network-centric (e.g. HTTP caches, proxies, file servers, routers) in the sense that they require high-performance network connections. Such performance is often achieved with application-specific specialization of system I/O [Cao95] requiring a modification of a portion of the operating system, such as the protocol stack or the file system.

Safety is a major concern for appliances, since new functions are constantly added, and there is never enough time to debug all possible interactions, especially when third party software is running on the appliance. The problem is more severe with legacy software written in C or other unsafe languages (in contrast with type-safe languages such as Java). Extensible routers are an example of network appliances where custom software processing has to be quickly added and safety is paramount. Diagnostic code and custom (e.g. multimedia) schedulers are also examples of extensions that appliances will need to support.

Resource management is especially important for appliances. New operating system abstractions such as *paths* [Mosberger96], *resource containers* [Banga99], *reservation domains* [Bruno98] and *activities* [Jones95] have been proposed for resource management and control. We believe that an appliance operating system should be flexible enough to support such abstractions. It should also be able to overlay resource management in modular operating systems without such support.

Pebble [Gabber99] is a component-based operating system that combines efficient IPC with strong modularity and safety features. Pebble provides the necessary infrastructure for building fine-grained, modular operating systems for appliances out of reusable components. We found that a typical network appliance application (e.g. a Web server) built on Pebble using components has comparable performance to a traditional monolithic kernel. In this way, there is no performance reason not to use a safer, more modular component structure for such appliances. Moreover, we claim that Pebble is a general purpose operating system framework that allows us to implement diverse OS structures and mechanisms, alleviating the need for writing specialized operating systems from scratch.

2. Challenges and Opportunities in Component-Based Systems

We believe that fine grain decomposition of system services allows easier specialization and customization of functionality. This is achieved by replacing generic components with specialized policies and mechanisms (e.g. protocol stacks). An added benefit is improved modularity by composing the system of smaller pieces with well-defined interfaces. Such building blocks can be reused. Strong fault isolation is necessary in appliances, as described above in the motivation section. Using hardware VM mechanisms to enforce fault-isolation by sandboxing each component within its own virtual address space is likely to increase memory system pressure [Chen93]. But this is unlikely to degrade performance in today's high-end embedded processors due to improved cache designs [Transmeta00]. Moreover, since most current high-end embedded processors (e.g. MIPS, StrongARM) include a hardware MMU, there is an incentive to use it.

* Contact author: MD 233, 33 Oxford Street, Cambridge MA 02138. email: magoutis@eecs.harvard.edu

We believe that interfaces between components need not be designed once and remain static or be painful to change. Rather, we think that interfaces should dynamically adapt and be optimized for the particular trust relationship and parameter passing between corresponding components. New applications may define new interfaces or modify existing ones as necessary to improve performance. For example, the spread of the World Wide Web and the focus on the performance of Web servers has highlighted shortcomings of the UNIX I/O system interfaces, where changes are notoriously hard to incorporate and standardize [Nahum99]. In another example, the continuous evolution of the internal Linux kernel interfaces does not allow interoperation with proprietary binary-only device drivers. This is because changes to the interfaces cannot be propagated into the drivers without rebuilding them from source.

One of the key challenges we encountered when building a fully component-based networking application in Pebble is how to efficiently share state between communicating components (such as network buffers). Several earlier studies suggested methods for improving the throughput in a system where data moves between multiple protection domains [Druschel93, Brustoloni98]. These methods rely on page remapping to avoid expensive data copying. But when the cost of data copying is comparable to the overhead of TLB operations, it may be preferable to dynamically select copying over page remapping.

3. Pebble Architecture Overview

Pebble [Gabber99] is a new operating system designed for flexibility, safety and high performance. Pebble supports composing an operating system of fine-grain components, each running within its own protection domain. Efficient inter-domain communication is achieved via a *portal*, an efficient IPC mechanism whose dynamically generated code is specified by a small interface definition language. Pebble provides a set of features that are especially useful for a network appliance operating system:

- **Dynamic portal specification, creation and optimization** allows for flexible interfaces and efficient communication between components. Components are able to dynamically define their interfaces, therefore being able to adapt. Portals are optimized for the parameter passing and trust relationship between the corresponding components. Portals copy small arguments and pass larger arguments (e.g., buffers) on a memory page remapped into the called protection domain. The dynamic nature of portal specification and creation is especially important in an extensible operating system. This is because dynamically loaded extensions frequently need changes to the existing API in order to support the new functionality.
- **Fine-grain modularity** is supported by efficient portals. It allows for finer system decomposition which makes customization easy. As a result, new functionality may be introduced by replacing a small number of components (possibly a single component) with little impact on other components. In this way, the system has a minimal footprint, since it is composed only of the set of necessary components. Pebble also supports coarser grain modularity, which is handy when the software is monolithic or when the components are trusting each other.
- **Protection** is provided by hardware-enforced virtual address spaces. It is essential for rapid deployment of new functionality written in any language (not just type-safe languages such as Java).
- **Thread migration** is the native mechanism for passing control between components. Pebble threads migrate from one protection domain to another by a portal call. No scheduling action (hence no context switch) is taken at the portal traversal. Threads are preempted either voluntarily, by blocking on a resource (semaphore), by waking up a higher priority thread, or by the interrupt dispatcher which wakes up a higher priority interrupt handling thread. Migrating threads simplify resource accounting for client/server activities, as described in the next section.
- **Interposition** of code in the portal interfaces between components can be used, for example, to add resource accounting and limit checking in modular systems.

Pebble primitives are very efficient. For example, a one-way IPC can be done in 135 cycles [Gabber99] on a MIPS R5000, using a new callee stack and opening a one page memory window.

Pebble further aims to improve on the existing model of extensibility offered by microkernels based on hardware VM mechanisms. In such systems, server components can be specialized for the use of a particular application, but it usually takes a disproportionate amount of effort to effect even small changes. We believe that alternative extensibility technologies (such as those based on type-safe languages or instruction-level techniques) can provide an orthogonal way to finely structure servers within Pebble components.

4. Implementing OS Services and Structures in Pebble

Some of the requirements from an appliance operating system are the ability for easy software updates, fast deployment of safe kernel functionality, higher fault-tolerance, as well as safe and controlled execution of third-party, untrusted code. Additionally, appliances should not rely on prompt administrator assistance for recovery, software updates, or execution of sophisticated diagnostic code. Finally, failure resiliency and defense against misbehaving extensions (e.g. denial of service attacks) both rely on resource management mechanisms to ensure consistent system state after crash or termination of a component.

Pebble can provide support for the above requirements. In this section we discuss (a) resource management abstractions; (b) component failure and termination; (c) the right fault isolation model and granularity; and (d) safe sharing of information.

4.1 Resource Management

We believe that Pebble can implement diverse resource management mechanisms, such as *paths* [Mosberger96], *resource containers* [Banga99], *reservation domains* [Bruno98] and *activities* [Jones95], without significant performance penalty due to a combination of architecture features that are described below.

(a) separating the notions of thread and resource principal using an entity similar to resource containers [Banga99]. This entity can be charged for resource use (e.g. by network connections, threads, protection domains such as protocols, applications, etc.);

(b) accurate resource accounting (CPU cycles, memory, I/O operations, etc.), which can be implemented by modifying the corresponding portals, since all system activities are done via portals. For example, physical memory allocation for a component's heap is done via the `sbrk()` portal call, CPU control is granted and relinquished via scheduler portal calls, other resources such as semaphores are represented by portals to the scheduler, and networking operations are all done via per-connection portals;

(c) providing the scheduler with additional information for scheduling multimedia activities. For example, queue operations need to be controlled by a trusted party and queue state be exported to the scheduler, so that this information can be safely used for thread deadline calculations.

Pebble is also suitable for adding resource management to existing modular applications that were not designed for it. Since most resource creation, transfer and use is implemented by portal calls, such calls may be efficiently intercepted by a management component that implements resource management [Gabber99].

4.2 Fault Tolerance

Pebble facilitates recovery from component failure and termination by recording portal calls that manipulate resources in an undo log. Since all external resources allocated by the failed component must be requested by portal calls, the recovery can release them rolling back the undo log. Earlier systems that employ migrating threads, such as Spring [Hamilton93], defined failure semantics that to a large extent apply to Pebble as well.

4.3 Choosing the Right Fault Isolation Model and Granularity

Pebble supports VM-based hardware fault isolation via protection domains. Some domains may run a Java virtual machine or employ various instruction-level techniques, such as software fault isolation (SFI) [Wahbe93]. We believe that using SFI for protecting modules is not a good idea in general, since SFI overhead can be prohibitive [Small96] (e.g. in the case of a Scout [Mosberger96] MPEG module). Nevertheless, SFI may be more efficient than hardware fault isolation for frequently executed fine-grain modules. In other words, SFI should be selected when its overhead is less than that of a portal call.

We plan to provide a new mechanism for communication between several SFI domains within a single Pebble component. This mechanism will switch between SFI domains by executing trusted code, similar to a portal call.

Good candidates for SFI are *packet classifiers,* which perform early demultiplexing of incoming packets. Packet classifiers are installed dynamically for filtering packets and are executed for each packet. In addition, packet classifiers are usually short, which means that they may be protected more efficiently by SFI than by hardware fault isolation.

On the issue of component granularity, one way to amortize the portal call cost is by batching requests. We are planning to experiment with a very fine-grained architecture similar to the Click modular router [Morris99] in order to investigate this idea.

4.4 Sharing Memory

Pebble can eliminate data copying by selectively sharing VM pages between components running in separate protection domains. Each shared memory page in Pebble has an access vector, which specifies the protection domains that can access it. The access vector enables diverse access manipulations, including sharing memory pages between protection domains, exchanging memory pages, allowing a server to access a page containing parameters only for the duration of the service call, etc. These mechanisms can be used for implementing both copy avoidance and clean separation of state. For example, we can implement the copy avoidance mechanisms of IO-Lite [Pai99] and emulated copy [Brustoloni99]. In addition, Pebble's shared memory pages are allocated from a single address space and as such they may contain pointers and other data structures in addition to data buffers.

5. Target Applications

This section describes potential applications that can benefit from Pebble mechanisms.

5.1 Adding Fault-Isolation to a Modular Router Architecture

A promising application area for network-centric appliances is differentiated services (*diffserv*) edge routers, which provide per-application traffic conditioning and other services at the border between a local area network and the Internet. Currently used traffic conditioners include elements such as meters, markers, shapers and droppers [Blake98]. Future needs may necessitate the adoption and deployment of services, which will require extensible router architectures. Current proposals for modular router architectures can be augmented with safety at the appropriate granularity using Pebble mechanisms.

Click [Morris99] is a flexible, modular software architecture for building routers. Click's building blocks are packet processing modules (called *elements*). Click's elements perform some routing functions, and they are connected into a graph. An example graph could be a *diffserv* edge router built by combining off-the-shelf traffic conditioner elements. Click's elements are usually fine grain. Most take between 22 to 135 machine cycles per packet on a 450 MHz Pentium II [Morris00]. Click has been originally implemented to run under Linux either as a user-level program or as a kernel extension.

Kernel extensions provide a method for trusted code to run within the kernel address space with no safety guarantees. We think that kernel extensions are unacceptable for an architecture such as Click, since new untrusted modules are expected to be developed and run in the kernel. Implementing Click on the Pebble infrastructure would add fault-isolation for elements, which will run in separate protection domains. However, we must reduce the domain call overhead by batching the processing of several packets in each element.

5.2 Support for Binary-only Modules

It is often the case that operating system components such as device drivers can only be provided in binary form due to licensing restrictions. However, a binary-only driver will fail once the interface to other kernel services is modified, which is a frequent event (e.g. in the Linux kernel[Usenix00]). Pebble accommodates changing interfaces by generating portals dynamically.

5.3 Automatic Diagnostics

Automatic diagnostics of performance or other problems in the field [Banga00] can be aided by execution of diagnostic modules written by the appliance manufacturer and ran safely on the appliance. The manufacturer often needs to quickly deploy new diagnostic code without waiting for the next release of the appliance operating system.

6. A Web Server Appliance

A component-based system is not necessarily slower than a functionally equivalent system with a more integrated structure as measured by end-to-end performance. We implemented a simple, single-threaded, event-driven Web server application using a TCP stack component specialized for HTTP traffic. The specializations aimed to reduce the per-connection latency (see [Nahum99]). We ran the Webstone benchmark to measure Web server performance on Pebble (with specialized TCP) and OpenBSD (with a standard TCP network stack). Pebble includes the following relevant components which run in separate protection domains: interrupt dispatcher, scheduler, ethernet driver, TCP protocol stack and application. We found that Pebble has comparable performance to OpenBSD, even for small transactions (64 to 2K bytes). Although Pebble has higher latency than OpenBSD (measurements show about 14% higher 1-byte UDP round-trip time compared to OpenBSD), we can easily specialize the TCP stack for HTTP on Pebble to eliminate four packets from a single HTTP GET or PUT operation. Such specializations are normally harder to do in a monolithic kernel.

7. Related Work

Scout [Mosberger96] is a communication-oriented operating system based on the *path* abstraction that helps optimize data flow between ends such as I/O devices. An instance of Scout targeted for a particular network appliance is statically generated from a collection of building-block modules with the resulting modular system optimized on a path-by-path basis. Pebble differs from Scout in several ways. First, Pebble's goal is to support a safer execution model where each system component runs within its own address space for fault-isolation. Second, Pebble's inter-domain communication is done via optimized, dynamically generated code. Scout is optimized at compile-time and runs in a single address space, while Pebble is geared towards run-time optimization and provides hardware-enforced protection within components. Escort [Spatscheck99], which adds multiple protection domains to Scout, has been found to have a high overhead.

Exokernel [Kaashoek97] is a radical approach to operating system design aiming to exterminate all high-level kernel abstractions (implemented by trusted servers in microkernels) and focusing on safely multiplexing low-level kernel resources. Exokernel implements high-level abstractions in user-space libraries (libOS's). Leaving all policy in user-space allows them to be easily specialized. Although an Exokernel and libOS implementation have been shown to be a good idea for certain applications [Kaashoek97], it is not yet clear how resources can be fairly shared among competing libOS's. Although libOS structure is orthogonal to the Exokernel architecture, published results [Kaashoek97] refer to a monolithic libOS. Building a network appliance for performance with such a system is bound to give rise to a number of problems when new appliances are added in the system.

8. Summary and Conclusions

We believe that an appliance operating system should be built from fine-grained components instead of building it from scratch for a particular use, or scaling down an existing operating system. A component-based system is inherently safer, while providing new services required by appliances, such as resource management, fault tolerance, flexible data sharing, support for binary-only modules and automatic diagnostics. The performance of a typical network appliance, a simple Web server running under Pebble, is comparable to an equivalent server running under OpenBSD, a mature monolithic system. We believe that there is no performance related reason not to use a safer, more modular component structure for such appliances.

9. References

[Banga99] G. Banga *et al.*, Resource Containers: A New Facility for Resource Management in Server Systems, in *Proceedings of the Third OSDI*, pages 45-58, February 1999.

[Banga00] G. Banga, Auto-diagnosis of Field Problems in an Appliance Operating System, in *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 293-306, June 2000.

[Blake98] S. Blake, et al., An Architecture for Differentiated Services, Network Working Group RFC 2475.

[Bruno98] J. Bruno, *et al*. The Eclipse Operating System: Providing Quality of Service via Reservation Domains, in *Proceedings of 1998 USENIX Technical Conference*, pages 235-246, June 1998.

[Brustoloni98] J. Brustoloni, P. Steenkiste. User-Level Protocol Servers with Kernel-Level Performance, in *Proceedings of the INFOCOM'98*, pages 463-471, March 1998.

[Brustoloni99] J. Brustoloni. Interoperation of Copy Avoidance in Network and File I/O, in *Proceedings of the INFOCOM'99*, pages 534-542, March 1999.

[CacheOS] CacheOS. http://www.cacheflow.com/technology/

[Cao95] Pei Cao *et al*. A Study of Integrated Prefetching and Caching Strategies, In *Proceedings of the SIGMETRICS'95*, pages 188-197, May 1995.

[Chen93] J. Chen and B.N. Bershad, The Impact of Operating System Structure on Memory System Performance, In *Proceedings of the 14th SOSP*, pages 120-133, December 1993.

[Druschel93] P. Druschel, L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility, in *Proceedings of the 14th SOSP*, pages 189-202, December 1993.

[Gabber99] E. Gabber *et al*. The Pebble Component-Based Operating System, In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 267-281, June 1999.

[Hamilton93] G. Hamilton, P. Kougiouris, The Spring Nucleus: A Microkernel for Objects, *SUN Microsystems Technical Report* SMLI TR-93-14, April 1993.

[Jaeger99] T. Jaeger *et al*. The SawMill Multiserver Approach, *Submitted to 9th ACM SIGOPS European Workshop, 1999*.

[Jones95] Jones, M.B., *et al.* Modular, Real-Time Resource Management in the Rialto Operating System. In *Proceedings of the 5th HotOS Workshop*, pages 12-17, May 1995.

[Kaashoek97] Kaashoek, M., *et al*. Application Performance and Flexibility in Exokernel Systems, in Proceedings of the 16th SOSP, pages 52-65, October 1997.

[Mosberger96] D. Mosberger, L. Peterson. Making Paths Explicit in the Scout Operating System, In *Proceedings of the Second OSDI*, pages 153-168, October 1996.

[Morris00] R. Morris. Personal communication.

[Morris99] R. Morris *et al*. The Click Modular Router, In *Proceedings of the 17th SOSP*, pages 217-231, December 1999.

[Nahum99] E. Nahum et al. Performance Issues in WWW Servers, *IBM Technical Report*, 1999

[Pai99] V. S. Pai, P. Druschel, W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System, in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 15-28, February 1999.

[Small96] C. Small, M. Seltzer. A Comparison of Operating System Extension Technologies, In *Proceedings of the USENIX'96 Annual Technical Conference*, pages 41-54, January 1996.

[Spatscheck99] O. Spatscheck, L. Peterson, Defending Against Denial of Service Attacks in Scout, In *Proceedings of the Third OSDI*, pages 59-72, February 1999.

[Sullivan00] D. Sullivan, M. Seltzer, Isolation with Flexibility: A Resource Management Framework for Central Servers, in *Proceedings of the 2000 USENIX Technical Conference*, pages 337-350, June 2000.

[Transmeta00] http://www.transmeta.com

[Usenix00] L. Torvalds, BOF Session at 2000 USENIX Technical Conference, June 2000.

[Wahbe93] Wahbe, R. *et al*. Efficient Software-based Fault Isolation, In *Proceedings of the 14th SOSP*, pages 203-216, December 1993.