A Microkernel API for Fine-Grained Decomposition

Sebastian Reichelt Jan Stoess Frank Bellosa
System Architecture Group, University of Karlsruhe, Germany
{reichelt,stoess,bellosa}@ira.uka.de

ABSTRACT

Microkernel-based operating systems typically require special attention to issues that otherwise arise only in distributed systems. The resulting extra code degrades performance and increases development effort, severely limiting decomposition granularity.

We present a new microkernel design that enables OS developers to decompose systems into very fine-grained servers. We avoid the typical obstacles by defining servers as lightweight, passive objects. We replace complex IPC mechanisms by a simple function-call approach, and our passive, module-like server model obviates the need to create threads in every server. Server code is compiled into small self-contained files, which can be loaded into the same address space (for speed) or different address spaces (for safety).

For evaluation, we have developed a kernel according to our design, and a networking-capable multi-server system on top. Each driver is a separate server, and the networking stack is split into individual layers. Benchmarks on IA-32 hardware indicate promising results regarding server granularity and performance.

1. INTRODUCTION

An operating system (OS) can be designed either as a monolithic kernel, or as a microkernel with individual servers running on top. While the technological benefits and challenges of microkernel-based systems have been explored in research and practice [15, 16], less attention has been spent on the programming environment they provide to the OS developer. Monolithic kernel programming is comparatively similar to regular application programming, albeit with special concerns for low-level issues. In contrast, a microkernel-based multi-server OS corresponds to a distributed system consisting of many individual programs, all of which must be programmed to interact correctly.

The similarity to distributed environments directly stems from the microkernel APIs in existence. The need, for instance, to explicitly pass messages between servers, or the need to set up threads and address spaces in every server for parallelism or protection require OS developers to adopt the mindset of a distributed-system programmer rather than to take advantage of their knowledge on traditional OS design.

Distributed-system paradigms, though well-understood and suited for physically (and, thus, coarsely) partitioned systems, present obstacles to the fine-grained decomposition required to exploit the benefits of microkernels: First, a lot of development effort must be spent into matching the OS structure to the architecture of the selected microkernel, which also hinders porting existing code from monolithic systems. Second, the more servers exist — a desired property from the viewpoint of fault containment — the more additional effort is required to manage their increasingly complex interaction. As a result, existing microkernel-based systems are typically restricted to fairly coarse-grained decomposition, lest development overhead and performance penalties render them unmaintainable or unusable.

We present an alternative microkernel design, which strives to overcome the limits to OS decomposition by replacing distributed-system concepts with a simple component model. Although most multi-server systems employ some component framework, it is usually built on top of an existing microkernel. Instead, we take the reverse direction: First, we design a component framework with fine-grained OS decomposition in mind, then we develop a microkernel that *directly* implements the framework. This way, we are able to keep the component model free of explicit, distributed-system-like concepts, and to handle isolation of components, concurrency, etc. on a more abstract level. Servers are not tasks but passive objects, whose code is executed in the context of their callers. They usually neither create any threads, nor wait for or send messages. Their interaction is formalized in a way that permits marshaling for calls across address spaces, but they can also be loaded into the same address space and/or executed in kernel mode.

To demonstrate our approach, we have developed a component model and a corresponding prototypical microkernel, along with a multi-server OS running atop. The multi-server OS includes an Ethernet driver ported from Linux and a network stack based on the lwIP project [8]. We successfully decomposed all drivers and network layers into individual servers. On average, we achieved a granularity of about 300

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLOS '09, October 11, 2009, Big Sky, Montana, USA.
Copyright 2009 ACM 978-1-60558-844-5/09/10...$10.00

lines of code per server. At present, we load all servers into a single address space and execute them in kernel mode. Benchmarks indicate that server invocation incurs an overhead of 2.5 to 4 times the duration of regular function calls. Reduction of this cost, as well as support for multiple address spaces, are targets of our ongoing work.

The rest of this paper is structured as follows: In section 2, we compare our approach to existing microkernels. In section 3, we explain the key aspects of our design. In section 4, we discuss our implementation and evaluation results.

2. RELATED WORK

The primary goal of microkernels is to separate OSes into independent servers that can be isolated from each other, while enabling the resulting multi-server systems to provide at least the same features as existing monolithic systems. Purported benefits include robustness and security due to containment of faults and malicious code, easier extensibility and maintainability by strict enforcement of roles and interfaces, as well as flexible verbatim server reuse in the face of changing system requirements [5, 13, 21, 25].

Early microkernels such as Mach implemented a stripped-down variant of a fully-fledged kernel, including drivers, file systems, process management, etc., aiming at gradually replacing some kernel features by user-mode daemon processes [1]. Complex abstractions, however, limit the performance of microkernel operations, and consequently the achievable system granularity. Later microkernels such as L4 and Exokernel reduced the number of abstractions and mechanisms to the bare minimum required for the implementation of secure and fast multi-server systems [9, 20], typically comprising: **i)** support for protection via address spaces, **ii)** execution of code in protection domains, and **iii)** communication and data sharing across protection boundaries.

These abstractions manifest themselves as additional concepts lacking any equivalent in monolithic kernels. Active, task-based architectures, as seen in early microkernels, are still the most common (and have also found their way into language-based approaches such as Singularity [2]). But even in systems without tasks (e.g., in Hydra [7] or Pebble [6]), there are other new concepts such as access rights, indirect manipulation of data, or explicit setup of communication paths. If the OS consists of a few large servers, as in L4Linux [15], ExOS [17], or virtual machine monitors [14], such additional microkernel-specific concepts can be largely ignored. However, developers of true multi-server systems, such as Workplace OS [23], Hurd [26], SawMill [13], or K42 [18], always face the problem of having to adapt every part of the system accordingly.

Existing attempts to keep the effort manageable include remote procedure calls (RPC) built on top of microkernel mechanisms [18, 24], support libraries for memory management or binding in servers [4], or advanced distributed-system concepts such as Clustered Objects [3]. However, hiding the microkernel behind a facade does not free OS developers from the need to tailor the system to microkernel paradigms at all times. For instance, the use of RPC reduces client code to simple function calls, but does not solve interaction and coordination problems when servers become clients themselves; nor does it help in splitting data structures across multiple servers. The case of SawMill shows that such bold semantic changes easily outweigh the potential benefits of a multi-server system. Particularly, every necessary step requires detailed knowledge of *both* microkernel *and* OS semantics.

Finally, in contrast to mere component frameworks such as OSKit [11] or Think [10], our model explicitly allows for component isolation, supporting all the benefits of microkernel-based systems by design.

3. DESIGNING FOR DECOMPOSITION

The core of our microkernel architecture is a specially crafted component model. In existing multi-server systems, servers can be regarded as components only to some extent; they are still tasks that consist of address spaces and threads, and interact by sending and receiving messages. Our servers, however, are components and nothing else. The component model is simple enough to be implemented directly by a microkernel, but also expressive enough for servers to be loaded into different protection domains.

Our microkernel's main purpose is to load components and manage their interaction according to the rules defined by the model. In contrast to usual APIs, which define a set of kernel features, our model tells developers how to write servers. These are compiled into tiny files, which the kernel can load and connect. The kernel's exact behavior is an implementation detail; it just needs to satisfy the rules of the component model. Specific kernel features, such as memory management, security, hardware access, etc., are not part of the component model itself, but can be defined on top.

Before writing a server, one needs to define an interface it will implement. Its function signatures are attached to the server in machine-readable form; there is no global interface registry. Figure 1 shows an interface of a hypothetical file system server, with a single function returning the root directory. A directory is also represented by an interface, as shown in Figure 2.

The component model defines three different parameter and return value types: **i)** scalars (up to machine word size), **ii)** references to interfaces (as returned by `getRootDirectory` in Figure 1), and **iii)** blocks of memory referenced by pointers, which are copied between the caller and callee in one or both directions. Such a parameter, as seen in the `getName`,

```
Directory getRootDirectory();
```

Figure 1: `FileSystem` interface.

```
Size getName(char *name, Size nameSize);
Directory getDirectory
            (const char *name, Size nameSize);
File getFile(const char *name, Size nameSize);
Iterator listDirectories();
Iterator listFiles();
```

Figure 2: `Directory` interface.

getDirectory, and getFile functions in Figure 2, is accompanied by an additional size parameter (and optionally a return value) specifying the amount of data to copy.

A server implementing the **FileSystem** interface will need to return an interface reference from getRootDirectory. It can obtain such a reference by loading another server implementing the **Directory** interface, or by calling a function (of some server) returning such a reference. In a file system, however, the root directory is fundamentally part of the same server. In fact, individual files and directories of a file system can never be regarded as isolated components, but are interrelated parts of a single data structure. In this (common) scenario, the server can construct a local object implementing an interface. It is able to access the data of this object via a locally valid pointer, but it also receives an interface reference, which it can pass to another server.

In addition to implementing an interface, a server can specify required interfaces. When the server is loaded, the entity loading it must provide a reference for each of these interfaces. For example, a file system server will specify an interface to a block device; loading the file system server is equivalent to mounting the file system on that device. Since all external dependencies have to be specified in this way, servers are self-contained; they do not have any global requirements. This property obviates the need for a general naming subsystem, as typically found in distributed systems.

So far, what we have described is merely a component model. Its only direct relation to a microkernel is that servers can be loaded into different address spaces due to the restrictions we placed on function signatures. These restrictions also ensure that the component model can feasibly be implemented directly, without any additional abstraction layer on top of the kernel. However, since the component model is meant to fully replace the microkernel API, it must encompass more of the features necessary for OS development; we describe the most important ones in the following paragraphs.

3.1 Concurrency

Existing microkernels handle concurrency by letting servers create multiple threads. This is very different from monolithic kernels, where modules can simply be used by multiple threads in parallel, provided that they use appropriate locking mechanisms internally. Our passive components work exactly like modules in a monolithic kernel: They can be declared as "thread-safe" and equipped with locks, and consequently will be called in parallel if two or more threads invoke the same servers concurrently.

In microkernel-based systems, such a threading architecture is referred to as a "migrating thread" model [12] (see Figure 3). In our case, however, it arises naturally from the passivity of the components. In other words, we did not apply any specific threading model, but merely avoided all microkernel-specific requirements to deal with threads in the first place. The result is equivalent to the model used implicitly in any component architecture that is not distributed. Thus, in our design, threading and isolation are orthogonal.

Having a foreign thread execute code of a server may seem to raise security issues. However, we can resolve them easily by

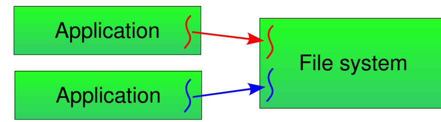

Figure 3: Passive servers and migrating threads enable implicit concurrency within servers.

ensuring that all threads entering a server are treated equally from the server's point of view. In fact, the same model is employed in virtually every monolithic kernel, when user-level code makes a system call: The thread that has been running at user level continues to execute kernel code. Our model simply extends the vertical structuring of monolithic kernel and user space to the multi-server system domain [19].

For the OS developer, making a server thread-safe in our system works very similarly to developing a thread-safe class in a high-level language such as Java: Just like individual methods of the class need to be protected against concurrent execution, locks must be placed around the bodies of server functions that are externally accessible via an interface. Since the kernel manages all incoming and outgoing calls, this can actually be automated in a straightforward manner, so that explicit locking code is usually not needed in the server. The kernel is free to use the most appropriate locking mechanism, or even omit the locks if it can guarantee that the locked regions are never executed concurrently. We tolerate the implications on kernel complexity if that enables us to eliminate the same complexity in server code, where we consider it much more harmful.

3.2 Protection

The restrictions we place on interface definitions ensure that servers can be isolated from each other by hardware means. Access control, i.e. management of the right to call other servers, is a separate issue, but does not require any new concepts in the component model: Since interface references are local handles managed by the microkernel, they can act as capabilities [22]. The only requirement on the kernel is that it manages them on a per-server basis, like UNIX file descriptors are managed on a per-process basis.

3.3 Data Sharing

Along with protection, most microkernels provide means to share data across protection boundaries, using the processor's memory management unit (MMU). The primary reason is performance: data that is shared does not need to be copied. In the presence of memory-mapped I/O and direct memory access (DMA), sharing virtual or physical memory with hardware devices can also become a necessity.

We use regular interface references to denote blocks of sharable memory, so that memory can be treated exactly like other resources in terms of protection. Mapping a memory block is a special feature of the component model (perhaps the first fundamentally microkernel-related aspect we describe). Similarly to the Unix mmap call (using an interface reference in place of the file descriptor), the mapping request results in a pointer that can be used to access the memory. In contrast to most microkernels, explicit address space management is not required.

	Minimum	Maximum	Average
LOC	27	1363	306
Bytes	280	13588	2907

Table 1: Server sizes in our prototype system.

References representing actual physical memory can be obtained from the kernel. However, to achieve the flexibility expected from microkernel memory management, we also permit the client to specify a reference to an arbitrary server implementing a certain interface. On a page fault, the kernel calls a function of this interface, passing the fault location and details as arguments. The target server (which represents the "virtual" memory block) then specifies another interface reference to redirect the access to. It, in turn, can either represent physical memory handed out by the kernel, or be a user-implemented server. Using this scheme, we are able to implement common memory management paradigms such as on-demand allocation and copy-on-write semantics.

3.4 Object Lifetime

Since a microkernel implementing our component model is required to manage references between servers, it knows when servers or individual objects are no longer referenced. Therefore, the component model mandates that unreferenced objects are destroyed automatically. This enables servers to return references to newly created objects without having to manage their lifetime. In distributed-system-like environments, object lifetime management can become a serious problem: For instance, a server that creates an object on behalf of another server typically needs to be notified when that server ceases to exist.

4. EVALUATION

We have developed a kernel that implements the component model on the IA-32 architecture, as well as a prototypical multi-server system. Drivers for standard PC hardware, PCI, and a Realtek RTL-8139 100 MBit/s Ethernet adapter are implemented as servers, the RTL-8139 driver being ported from Linux. Furthermore, we have built a TCP/IP stack using code from the lwIP project [8], but separated into individual servers per networking layer.

An analysis of the source code shows that we are able to achieve a consistently fine server granularity (see Table 1). First of all, this indicates that the restrictions we place on interfaces between servers do not impact the ability to separate servers where appropriate. Second, it is a level of granularity that is difficult to achieve in a system based on tasks, because part of the code will always be related to additional issues such as starting and stopping, communication, etc.

Furthermore, we were able to achieve good results *reusing* individual pieces of code, particularly in the network device driver we ported. Figure 4 shows the amount of unmodified, adapted, and new code in the resulting server. 75 per cent of the code was reusable either without modification or with an often straightforward adaption to the server model (for example, the use of specific interfaces instead of direct calls to kernel subsystems, or the conversion from pointers to interface references). The largest change was the introduction of queues instead of immediate packet processing,

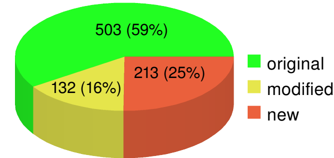

Figure 4: Reuse of code in the RTL-8139 driver.

which was necessary to achieve satisfactory performance in an arbitrary microkernel environment.

We performed micro-benchmarks on two Intel processors, a Pentium 4 and a Core 2 (see Table 2). Since our prototype kernel loads all servers into the kernel, a call from one server to another only takes about 2.5 to 4 times as long as a regular function call. The additional cost stems from the need to build additional stack frames containing information about the called servers, in order to satisfy core properties of the component model such as the lifetime of references. The overhead is kept low by dynamically generating custom code for each server function at load time, as seen in Pebble [6]. Another frequent server operation is obtaining and releasing a reference, e.g. if referenced objects are individual network packets. The duration in the prototype implementation is in the same range as a call between servers.

Although we have not yet implemented a complete microkernel with user-mode server support, we can predict the potential performance of calls across address spaces ("inter-AS") by setting up two address spaces with appropriate contents and kernel data structures, and implementing the necessary part of the kernel call/return path. This includes all required run-time checks and data structure modifications. The results are promising: A call and return between address spaces is faster than two IPCs on the L4Ka μ-kernel, which is explicitly designed for good IPC performance [15].

As a more real-world benchmark, we measured network data transfer performance using our ported (and decomposed) TCP/IP stack and Ethernet adapter driver, and compared it against the performance of the same card under Linux. We were able to achieve equal throughput, but the CPU load turned out to be approximately twice as large. This number needs to be taken with a grain of salt, however, since the TCP/IP stack we ported is very different from the highly optimized Linux equivalent.

5. CONCLUSION

We presented a microkernel API specifically designed for fine-grained decomposition, in the form of a specialized component architecture. We discarded distributed-system

	Pentium 4	Core 2
Kernel-mode call/return	46	23
Reference management	60	31
(Function call/return)	11	9
Inter-AS call/return	1965	771
(L4Ka round-trip IPC)	2262	1052

Table 2: Number of cycles of kernel operations.

paradigms in favor of passive objects communicating via function calls. Restrictions on the function signatures ensure that we are able to load servers into different address spaces and manage their communication.

To evaluate the feasibility and costs of fine-grained servers, we developed a prototype kernel and multi-server OS. We succeeded in separating all internal system components, individual drivers, and even different layers of a TCP/IP stack into servers. Our servers average around 300 lines of code.

Communication across servers does carry a performance cost compared to regular function calls, even if no address space switches are involved. The overhead is much lower than the cost of hardware address space switches, which implies that being able to load servers into the same address space is a worthwhile feature. However, for fine-grained systems that require a lot of server communication, the overhead can still become significant. Better performance is a goal of our ongoing work.

6. REFERENCES

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[2] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 1–10, San Jose, CA, 2006.

[3] J. Appavoo. *Clustered objects*. PhD thesis, University of Toronto, Ontario, Canada, 2005.

[4] M. Aron, L. Deller, K. Elphinstone, T. Jaeger, J. Liedtke, and Y. Park. The SawMill framework for virtual memory diversity. In *Proceedings of the 6th Asia-Pacific Computer Systems Architecture Conference*, Bond University, Gold Coast, QLD, Australia, Jan. 2001.

[5] B. N. Bershad, C. Chambers, D. Becker, E. G. Sirer, M. Fiuczynski, S. Savage, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[6] J. Bruno, J. Brustoloni, E. Gabber, A. Silberschatz, and C. Small. Pebble: A component-based operating systems for embedded applications. In *Proceedings of the USENIX Workshop on Embedded Systems*, pages 55–65, Cambridge, MA, Mar. 29–31 1999.

[7] E. Cohen and D. Jefferson. Protection in the Hydra operating system. In *Proceedings of the 5th Symposium on Operating System Principles*, pages 141–160, Austin, TX, 1975.

[8] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of the 1st International Conference on Mobile Applications, Systems and Services*, San Francisco, CA, May 2003.

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[10] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A software framework for component-based operating system kernels. In *Proceedings of the USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.

[11] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 38–51, Saint Malo, France, Oct. 5–8 1997.

[12] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating threads model. In *Proceedings of the Winter USENIX Conference*, San Francisco, CA, Jan. 1994.

[13] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, Sept.17–20 2000.

[14] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, June 2005.

[15] H. Härtig, M. Hohmuth, J. Liedtke, and S. Schönberg. The performance of μ-kernel based systems. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 66–77, Saint Malo, France, Oct. 5–8 1997.

[16] G. Heiser. Secure embedded systems need microkernels. *;login: the USENIX Association newsletter*, 30(6), Dec. 2005.

[17] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 66–77, Saint Malo, France, Oct. 5–8 1997.

[18] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *Proceedings of the 1st ACM SIGOPS EuroSys conference*, Leuven, Belgium, Apr. 2006.

[19] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, Sept. 1996.

[20] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 175–188, Asheville, NC, Dec. 5–8 1993.

[21] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 237–250, Copper Mountain, CO, Dec. 3–6 1995.

[22] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Systems Research Laboratory, Department of Computer Science, Johns Hopkins University, Mar. 2003.

[23] F. L. Rawson III. Experience with the development of a microkernel-based, multi-server operating system. In *Proceedings of 6th Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, May 5–6 1997.

[24] L. Reuther, V. Uhlig, and R. Aigner. Component interfaces in a microkernel-based system. In *Proceedings of the Third Workshop on System Design Automation (SDA)*, Rathen, Germany, Mar. 2000.

[25] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can We Make Operating Systems Reliable and Secure? *IEEE Computer*, 39(5):44–51, May 2006.

[26] N. H. Walfield and M. Brinkmann. A critique of the GNU Hurd multi-server operating system. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, July 2007.