Improving Dependability by Revisiting Operating System Design

Francis M. David, Jeffrey C. Carlyle, Ellick M. Chan, Philip A. Reames, Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N Goodwin Ave, Urbana, IL 61801
{fdavid,jcarlyle,emchan,reames,rhc}@uiuc.edu

Abstract

Existing operating system (OS) designs provide inadequate isolation of user applications from errors that occur in OS services. If an error causes the failure of an OS service, all dependent applications are affected. The OS design described in this paper ameliorates this problem by reorganizing OS state in an effort to make OS services transparently restartable. This is achieved by partitioning application-related OS state into isolated per-application memory regions. Access to these memory regions is provided to OS services on a "need-to-know" basis when processing application requests. Applications are not allowed access to these memory regions for security. This design helps improve the dependability of the system.

1 Introduction

Computer system reliability is an important concern in today's world and has fueled a significant amount of recent research into operating system (OS) structure [21, 16, 22]. Existing OS designs, however, do not adequately protect applications from runtime errors that occur inside the OS. These errors can be caused by buggy drivers [9] or by hardware faults. Errors which occur inside monolithic kernels can potentially corrupt both OS and application data. Microkernel designs partition the OS into smaller components that are mutually isolated. This organization helps to contain error propagation and improves the overall reliability of the system. OS services running on microkernels, however, usually maintain state corresponding to applications as illustrated in figure 1. This creates a tight coupling between applications and the OS service. For example, if the filesystem server in a microkernel such as Minix3 [14] is terminated because of an error, all running applications that depend on the filesystem also fail. The Minix3 reincarnation server simply restarts the failed service and does not help recover existing applications [13]. This shortcoming

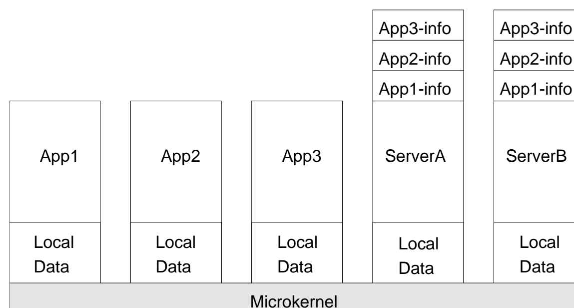

Figure 1. Microkernel OS structure

also exists in other microkernels like L4 [17], Chorus [18], and EROS [20]. While stateless OS services can be safely restarted, any server which holds application state presents a serious impediment to system reliability.

Our goal is to decouple an OS service from applications so that an error in the service does not affect all dependent applications. This is achieved by relocating application-specific state from the OS servers into special per-application memory regions referred to as Server State Regions (SSRs). This is illustrated in figure 2. In the figure,

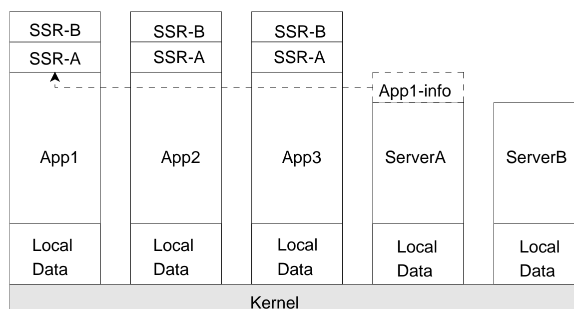

Figure 2. Proposed OS structure

"App1-info" (originally shown in figure 1), which represents state information about App1 maintained by ServerA, has been moved from ServerA to App1's SSR-A. With this design, error propagation can be minimized by controlling access to the SSR.

When an application requests an OS service, the kernel grants the server temporary access to the SSR. The server can only manipulate data in the SSR while processing the request. When the server completes processing the request, its permissions to the SSR are revoked. A multi-threaded server can have several SSRs corresponding to different client applications mapped in at the same time. For security, SSRs are inaccessible from applications. The kernel can use memory protection mechanisms to manage application and server permissions.

If an error occurs during the processing of a request, it is confined to the server's address space and therefore only affects the SSRs that are currently mapped in. When an error is detected, the server is immediately restarted. The kernel preserves SSRs through a restart. After restarting, the server is responsible for detecting corrupted SSRs and repairing them. If corrupted SSRs can not be transparently repaired, error codes are returned for pending requests. The restarted server seamlessly resumes servicing those applications whose SSRs were not corrupted. Thus, in our design, only a small subset of the dependent applications are affected. We assume that a server restart eliminates the error condition and returns the system to a usable state. This technique is similar in nature to a micro-reboot [5].

In a filesystem server, for example, if an error occurs when a corrupted disk block is parsed, only the application that was reading that file is affected. As soon as the filesystem server recovers from the error by restarting, all other applications can continue to access the filesystem normally.

The remainder of this paper is organized as follows. We discuss the management of state in section 2. Section 3 presents the design of some typical OS services that use SSRs. The dependability characteristics of our proposed OS structure are presented in section 4. Some additional aspects of our design are discussed in section 5. We compare our design with some related work in section 6 and conclude in section 7.

2 State Management

Our design attempts to protect client-related OS service state from errors that occur in the OS. It achieves this by partitioning this state into individual client-specific objects and distributing these objects to the clients. These objects are stored in SSRs associated with clients. For example, a storage server that provides access to raw disk blocks can store information about outstanding requests such as priority and requested blocks in client SSRs.

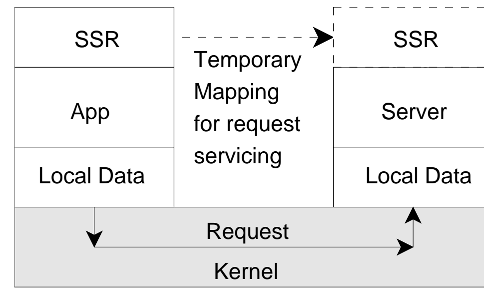

Figure 3. Servicing a Client Request

In our design a server can also be a client to other servers. For example, the filesystem server may be a client to the storage server. A memory paging server that provides disk-backed memory pages can, in turn, be a client to the storage server as well. It is possible to architect an entire OS environment using such restartable fault-isolated components.

An SSR is created when a client first binds to a server, and a new SSR is created for every server to which a client binds. While SSRs are created from the client's memory pool, the kernel sets the memory permissions on the SSR so that it cannot be accessed by code running in the client. This access control is required for security since it would otherwise be possible for the client to manipulate data objects used by the server.

SSRs are destroyed when clients unbind from a server. If a client is abruptly terminated, the kernel notifies servers whose SSRs are still associated with the client. Servers can use this notification to perform any cleanup operations that are necessary.

When a client binds to a server and initiates a service request, the kernel enters a mapping for the SSR into the virtual memory translation tables of the server before dispatching the request to it. This is illustrated in figure 3. The server can now access the SSR and use it to store and manage any client-specific state required to handle the request. When the server sends a response back to the process, the kernel removes the server's access to the SSR. Servers can be multi-threaded and have multiple client SSRs corresponding to in-progress client requests mapped in simultaneously. In this case, there is a trade-off in terms of the absence of fault isolation between simultaneously mapped in SSRs.

Some servers can be designed so that they are stateless and only require state information stored in the current SSR in order to process a request. For example, consider a server that encapsulates a sound card audio output device driver. Information such as the audio bit-rate can be stored in client SSRs and this is sufficient to process client requests. Some servers cannot be completely stateless and might require the maintenance of a global view of all client state in order to

process a request. In the storage service example, in addition to the state maintained in the SSRs, the server would need to maintain a queue of pending requests in order of priority. This global view can be maintained in the server address space as a cache of objects in the client SSRs. This cache can be maintained and verified during the processing of client requests.

A server can attempt to recover from a failure by restarting. Since SSRs are persistent across server restarts, a restarted server can continue to process requests from existing clients using the information stored in the SSRs. A restarted server first obtains a list of all related SSRs from the kernel and verifies their validity. A server specific recovery routine attempts to recover invalid SSRs. If an SSR cannot be recovered, an error is reported to the client that owns the SSR. Our isolation design ensures that only the SSRs that were used by the server after the occurrence of the error are affected by it.

If the sever requires a global view of all the SSRs, the cached objects can be re-created during recovery by obtaining all related SSRs from the kernel. For example, if the storage server crashes, the order of items in the request queue and job information stored in the server is lost. When it is restarted, it can reconstruct the job information and a new queue order from its SSRs present in clients. This recovery would be completely transparent to clients: they would not need to resubmit their disk block requests.

3 OS Service Examples

Our proposal requires designing and constructing OS services so that they use SSRs to manage client-specific state. In the previous section, we described how a storage service can be designed to use SSRs. In this section, we will briefly outline the design for a timer service, a filesystem service and a network service that use SSRs.

Timer Service: Operating systems usually include a timer service which can be used for periodic notifications. Clients present an interval period when they register with the service. A timer server can be designed to store this information in client SSRs. The service also maintains a local queue of all pending notifications that it uses to service its clients. If the server dies and restarts, the queue of pending notifications is lost. The queue can be re-created with the information in all the client SSRs. When reconstructing the queue, the server can assume that the start time for all client periods is the current time. This ensures that clients continue to receive notifications albeit with a skew due to the queue re-creation. The worst possible skew corresponds to an entire period (or one missed notification). This happens when the server fails just before a notification was due.

Filesystem Service: A filesystem server that supports a POSIX API provides clients with file handles to access a file when it is in use. A server can store the mappings between file handles and filesystem inodes in the SSR for a client. Additional information such as the current read/write offset (index) in the file can also be stored in the SSR. These details are client-specific and can be used to provide uninterrupted access to files even if the server is restarted.

After a restart, system specific configuration settings such as disk partition information may be reloaded from a configuration manager service. The objects managing the buffer cache can be made to persist across the server restart or the cache can be recovered through other techniques such as those used in Rio [7].

Locks which are used to synchronize access to files can exist in the local state of the filesystem server. SSRs can hold redundant information about file locks held by clients. When the server fails and restarts, it can re-create internal lock state from this information in the SSRs.

Network Service: A network service that provides TCP or UDP connections can use SSRs to store client-related information such as socket information and port numbers. A server local copy can be maintained in order to process incoming packets and for queuing reasons. Buffers for packets may be maintained in the server.

If the server restarts after a failure, it can re-create all internal state using the information in SSRs. System-wide configuration settings such as the IP address and routing information may be loaded from files or a configuration manager service. It can then resume processing outgoing and incoming packets. Loss of packets when the server crashes is similar to packet loss on the network and can be recovered by higher level protocols. An alternative strategy may be to recover buffers using techniques similar to those used for the filesystem buffer cache.

4 Dependability Characteristics

In this section, we examine our design with respect to some of the dependability attributes presented in Avizienis et al [2].

Reliability: In our design, an error in a server can affect only those clients with which it has communicated until the error is detected and fixed by restarting the server. The SSRs of these clients may be potentially corrupted, but all other SSRs are guaranteed to be free of corruption. When the server is restarted, the clients with valid SSRs can continue to interact normally with it. This transparent server recovery improves the overall reliability of the system.

Availability: Since the SSR is allocated out of the client process' assigned memory pool, the client process cannot perform a denial of service attack by issuing requests that result in the allocation of large amounts of OS state. This helps ensure the availability of OS services.

Existing designs such as the reincarnation server in Minix3 ensure high availability by simply restarting managed servers when they fail. We take advantage of this technique to improve server availability in our system as well.

While our initial goal was to ensure that an error in the server did not unnecessarily affect clients, our design also helps with creating robust client applications. A client can request that its SSRs be persisted across client failures and restarts. Since the error that caused the failure cannot corrupt the SSR, it can be used to resume an interrupted session if the client can re-create its internal state using other techniques such as persistent memory. For example, if the SSR holds TCP connection information from the network server, the TCP connection can be resumed without interruption after a crash and restart of the client. This can improve both the availability and reliability of the client.

Confidentiality and Integrity: Our design follows the "Need-to-Know" security principle. A server only has access to SSRs for clients which it is currently servicing. The server is denied access to SSRs of clients which have bound to the server but do not have pending requests. Therefore, a subverted server can only compromise the confidentiality and integrity of data of clients that initiate requests after the server has been compromised.

We are currently working on a framework to check the integrity of an OS by verifying the integrity of individual objects in the system. SSRs can be associated with server specific checkers that can verify the integrity of the stored data objects. Encapsulating client-related state in SSRs makes it easier to identify clients that have corrupted SSRs and notify them.

Maintainability: Our design also improves the maintainability of the operating system. Since servers are designed to be restartable, server upgrades are a simple matter of terminating the old server and starting a new server. As long as the new server can interpret existing SSRs for the service, clients would not be affected and there would be no interruption of service during the upgrade.

5 Discussion

We believe that separating out client-related state from OS services in order to achieve our goals is not a difficult task. The required state can be easily identified from an analysis of data needed for server recovery. For services that cannot be made completely stateless, a local redundant copy of the SSR state can be maintained as described in section 2. The redundancy also helps in detecting errors. If the copies are inconsistent, an error can be signaled.

Compared to traditional OS server designs, a small increase in the complexity of server code is unavoidable when using our approach. We feel that this is a reasonable price to pay for increased dependability. Our design also incurs performance overheads due to the need to switch address spaces multiple times within the OS when processing some requests. This overhead is experienced by all microkernel systems. Our current implementation approach is to make use of a single address space with distinct protection domains for OS servers in a manner similar to Opal [6] to reduce this overhead. Additional performance improvements may be possible by exploiting architectural features such as tags on memory pages. On the ARM processor, this is supported by the "domain" concept.

Our design does not guarantee complete recovery from all OS errors. While error propagation is minimized by componentizing the OS into services, errors can still propagate through the poorly designed interfaces and result in failures of other components. Such cascading failures may result in an unusable system. The chances of successful recovery are significantly dependent on error detection latencies. If an error is detected soon after it occurs, the damage it can do is limited and the chances of successful recovery are high. Thus, additional improvements in reliability may be obtained by combining our design with techniques for early error detection such as SafeDrive [25]. OS lockup errors can also be detected using watchdog timers [10]. The only error detection mechanism available in a direct implementation of our design is virtual memory protection that signals invalid memory access errors.

SSRs can be implemented as objects representing memory pages so that the kernel can easily manage server and client access to the SSRs through virtual memory permissions in the page tables. Allocating SSRs at page granularities may result in inefficient memory usage. This can be addressed by either allocating multiple SSRs within the same page at the expense of reduced isolation, or by using future architectural support such as Mondriaan Memory Protection [24].

SSR metadata is maintained within the kernel as a table associated with process objects. This ensures the security of the SSRs and allows the kernel to easily map SSRs into server address spaces when processing a request.

In addition to the dependability characteristics discussed in the previous section, we believe that our OS organization affords several benefits over existing designs which couple client state along with server state. Stateless servers are likely to improve scalability because they can be easily replicated onto many processors. This is especially signif-

icant for future multi-core processor designs. The explicit separation of client-related OS state can provide simple designs and easy implementations of process checkpointing and migration algorithms. In order to do this in a traditional microkernel, process related OS state information has to be collected from multiple servers that provide system services.

6 Related Work

The idea of a stateless server is reminiscent of the original implementation of Sun's Network File System (NFS) [19]. One of the reasons for the NFS design is that it allows for easier recovery in the event of a server being restarted or a temporary failure in communication. Unfortunately, the stateless behavior of the NFS protocol left it vulnerable to attack [23, 15]. Our design avoids the forged credential attacks that plagued NFS because SSRs are managed by the kernel and cannot be forged by clients.

SSR design is different from shared memory. Unlike shared memory, clients are oblivious to the existence of SSRs. Persistent (across a server restart) memory segments can be used to implement SSRs entirely within a server, but this places the responsibility of protecting individual SSRs in the server. This also results in reduced SSR security and eliminates the option of charging SSR memory costs to clients.

Chorus [18] includes support for persistent memory that can be used to store and recover state after a "Hot-Restart" [1]. But this is not used by the OS and is provided as a service for applications. Also, these persistent memory regions are not protected from corruption due to server errors. Minix3 [14] also provides the ability to persist state across a restart using a data store server. As with Chorus, this functionality is not exploited by the OS.

SSR access control is different from passing object capabilities between clients and servers. Unlike capabilities, SSR access is only granted to predefined servers and is easily revoked. Also, when an SSR is mapped from a client to a server, the privileges to the SSR are escalated. This is not possible with many current implementations of capabilities.

Checkpointing is widely used to recover from software failures. The use of OS server checkpoints at every request is not a feasible solution because of runtime memory copying and kernel interaction overheads. There is also an uncertainty in the number of checkpoints required to roll back to a correct state. Additionally, all work done since the last checkpoint is lost. EROS [20], which uses complete system checkpoints to recover from a crash, avoids performance problems by running the checkpointing task in the background. But it is prone to the other mentioned issues with checkpointing.

The Fluke [11] and Cache [8] microkernels allow user processes to access and modify associated kernel objects. This is possible through a design that clearly defines and exports process related kernel state. While this decouples user processes from the microkernel and can allow for easy checkpointing and process migration implementations, it does not address the coupling of user processes with OS services such as the filesystem.

The QuickSilver system [12] uses transactions to recover to a consistent system state after a failure. But server failures are not recovered and clients receive error codes when they try to communicate with a failed server. Server replication is one approach used to address this issue. Replication, however, does not help recover from errors due to software bugs which affect all replicas. In our design, servers recover transparently to clients and do not require replication. There is a possibility that restarted servers do not re-encounter the bug that caused the failure [5]. Quicksilver also provides a log manager interface that can be used by servers to store data required for recovery. But, there is no mechanism that allows for isolation of per-client state. Also, logging encounters substantially more overheads than our lightweight memory isolation approach.

Nooks [21] increases the reliability of Linux by isolating faults in device drivers using virtual memory protection and by recovering failed drivers. Isolation of Linux device drivers in separate virtual machines has also been explored using the L4 microkernel [16]. OKE [3] isolates untrusted driver code but requires the use of a special safety enforcing compiler. Unlike our approach, these techniques do not isolate and protect client-related OS service state. However, the driver recovery techniques employed in these projects are also applicable to our design. SafeDrive [25] uses language based techniques to improve OS reliability by detecting type safety violations. Such techniques complement our approach and further improve the reliability of our design.

7 Concluding Remarks

We have proposed a technique to provide isolation to client-related OS service state so that when an OS server encounters an error, the propagation of the error is limited to active clients. Additionally, persistence of client-related state in SSRs allows servers to recover from failures through simple restarts.

We have started implementing our design using the Choices OS [4]. Since submission of this paper, a framework that supports state management using SSRs has been built and several OS services have been rewritten using this framework. We intend to publish an analysis of our implementation in future papers. Latest information on this project (CuriOS) can be found on our website at `http://choices.cs.uiuc.edu`.

Acknowledgments

We would like to thank the anonymous reviewers for insightful feedback that helped shape the final version of this paper. Part of this research was made possible by grants from DoCoMo Labs USA and generous support from Texas Instruments and Motorola.

References

[1] V. Abrossimov and F. Hemann. Fast Error Recovery in CHORUS/OS: The Hot-Restart Technology. Technical Report CSI-T4-96-34, Chorus Systems, Inc., August 1996.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[3] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *IEEE Open Architectures and Network Programming*, pages 141–152, 2002.

[4] R. H. Campbell, G. M. Johnston, and V. Russo. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)". *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. In *Symposium on Operating Systems Design and Implementation*, pages 31–44, San Francisco, CA, December 2004.

[6] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single Address Space Operating System. *ACM Trans. Comput. Syst.*, 12(4):271–307, 1994.

[7] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.

[8] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the first Symposium on Operating Systems Design and Implementation (OSDI)*, pages 179–193. USENIX Association, Nov. 1994.

[9] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An Empirical Study of Operating System Errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.

[10] F. M. David, J. C. Carlyle, and R. H. Campbell. Exploring Recovery from Operating System Lockups. In *USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[11] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and Execution Models in the Fluke Kernel. In *Proceedings of the third Symposium on Operating Systems Design and Implementation*, pages 101–115, New Orleans, LA, USA, 1999.

[12] R. Haskin, Y. Malachi, and G. Chan. Recovery Management in QuickSilver. *ACM Trans. Comput. Syst.*, 6(1):82–108, 1988.

[13] J. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Roadmap to a Failure-Resilient Operating System. *USENIX ;login*, 32:14–20, February 2007.

[14] J. N. Herder. Towards a True Microkernel Operating System. Master's thesis, Vrije Universiteit Amsterdam, 2005.

[15] R. Kennell and L. H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of the twelfth USENIX Security Symposium*, pages 295–308, 2003.

[16] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the sixth Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[17] J. Liedtke. On μ-Kernel Construction. In *Proceedings of the fifteenth Symposium on Operating Systems Principles*, pages 237–250, New York, NY, USA, 1995. ACM Press.

[18] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus Distributed Operating System. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA), 1992.

[19] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland, OR, USA, 1985.

[20] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

[21] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In *Symposium on Operating Systems Design and Implementation*, pages 1–16, 2004.

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

[23] W. Venema. Murphy's law and computer security. *Proceedings of the sixth USENIX Security Symposium*, page 187, 1996.

[24] E. Witchel, J. Cates, and K. Asanović. Mondrian Memory Protection. In *ASPLOS-X: Proceedings of the tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, New York, NY, USA, 2002. ACM Press.

[25] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harre, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Symposium on Operating Systems Design and Implementation*, Nov 2006.