Anonymous RPC: Low-Latency Protection in a 64-Bit Address Space

Curtis Yarvin, Richard Bukowski, and Thomas Anderson

Computer Science Division
University of California at Berkeley

Abstract

In this paper, we propose a method of reducing the latency of cross-domain remote procedure call (RPC). Traditional systems use separate address spaces to provide memory protection between separate processes, but even with a highly optimized RPC system, the cost of switching between address spaces can make cross-domain RPC's prohibitively expensive.

Our approach is to use *anonymity* instead of hardware page tables for protection. Logically independent memory segments are placed at random locations in the same address space and protection domain. With 64-bit virtual addresses, it is unlikely that a process will be able to locate any other segment by accidental or malicious memory probes; it impossible to corrupt a segment without knowing its location. The benefit is that a cross-domain RPC need not involve a hardware context switch. Measurements of our prototype implementation show that a round-trip null RPC takes only 7.7μs on an Intel 486-33.

1 Introduction

A traditional function of operating systems is providing *protection domains*, or areas of memory accessible only by the process which owns them. UNIX, for example, keeps every process in an entirely separate address space; other systems use a shared address space, but have a different page protection map for each process. Either way, keeping separate processes in separate protection domains provides *safety* and *security*: protection from buggy processes which accidentally touch memory locations they don't own, and protection from malicious processes trying to read or alter the memory of other processes. Safety and security are necessary in any modern operating system.

Using virtual memory hardware to enforce protection is flexible and powerful. However, it is also expensive. Giving each process its own address space increases the amount of context-specific state, and thus the cost of context switches. Context-switch cost contributes little to system overhead on normal UNIX systems; but it sets a strong lower bound on the cost of IPC. At a minimum, two context switches are required for each round-trip interprocess communication. This can limit the feasibility of splitting logically independent, but closely cooperating, modules into separate protection domains.

Software overhead in context switching once dominated the hardware cost. But, as Anderson et al. [1991] discuss, the former has decreased with processor improvements while the latter has not. The result is that context-switch times for conventionally protected systems have remained static and large, and now pose a significant impediment to extremely fine-grained IPC.

One solution is lightweight threads which share a single protection domain. Thread switching and communication is fast, but threads are not safe or secure, and even when used in a trusted environment are vulnerable to byzantine memory-corruption bugs.

This work was supported in part by the National Science Foundation (CDA-8722788), the Digital Equipment Corporation (the Systems Research Center and the External Research Program), and the AT&T Foundation. Yarvin was supported by a California MICRO Fellowship, Bukowski by an NSF Graduate Fellowship, and Anderson by a National Science Foundation Young Investigator Award.

Our goal is to build a system that combines the protection of UNIX processes and the speed of threads, to make IPC cheap enough to for heavy use. This could allow application interactions of a much finer grain than are now feasible, and large software systems could be structured as groups of cooperating processes instead of single monolithic entities.

The key to our approach is Druschel and Peterson's observation [Druschel & Peterson 1992] that, in a very large, sparse address space, virtual address mappings can act as *capabilities* [Dennis & Van Horn 1966]. If a process knows a segment's position in its address space, accessing it is trivial; without this knowledge, access is impossible. Protection can be accomplished by restricting the knowledge of segment mappings. We refer to this approach as *anonymity*.

Anonymity was not feasible before the advent of 64-bit architectures. A 32-bit address space is small enough that every valid page in the address space can be easily found through exhaustive search. But searching a 64-bit address space in this way is nontrivial. Thus, on a 64-bit machine, it is possible to randomly map unprotected pages in a shared region and use their virtual addresses as capabilities. This provides a fast and flexible way of sharing memory.

Druschel and Peterson use their approach to pass buffers between user-level protocol layers in their x-kernel system. We believe that anonymity can be used as a general-purpose protection mechanism, providing reasonable safety and security to independent processes sharing the same address space. The protection this provides is *probabilistic*, but effective. Between processes in the same address space, a context switch is a simple matter of swapping registers and stacks, and can be performed with the same efficiency as with lightweight threads.

We have developed a simple prototype of anonymous protection; our implementation can perform a round-trip null procedure call between two protected domains in only 7.7μs on an Intel 486-33.

The rest of this paper discusses these topics in more detail. Section 2 shows how we can use a large address space to implement probabilistic protection; Section 3 shows how we preserve anonymity during cross-domain communication. Section 4 outlines some potential uses of anonymous protection, while Section 5 outlines some limitations of our approach. Section 6 presents performance results; Section 7 considers related work. Section 8 summarizes our experiences.

2 Implementing Anonymous Protection

Safe and secure anonymity is not difficult to implement, but requires some care.

First we need a way to assign addresses. When a segment (any piece of memory that must be mapped contiguously) is loaded, it needs a virtual address. If the segment is to remain anonymous, no process that is not explicitly given this address may be allowed to discover or derive it.

So we must select the address randomly. If the address is truly random, than no better algorithm exists to discover it than brute-force search. Unfortunately, there is no such thing as a truly random number generator. The best we can do is a cryptosystem, such as DES [Nat 1977]; if the key is a secure password or code, and the plaintext an allocation sequence number, then the resulting encrypted text will be securely random. We transform the encrypted text into a virtual address and map the segment at that position (unless it would overlap another segment, in which case we compute another address).

This ensures that the most efficient algorithm for finding other processes' data will be brute-force search: iterating through virtual space and dereferencing every page (or every segment width, if segments are larger than pages), intentionally ignoring segmentation faults until a valid page is found. (The UNIX signal handling machinery, for example, allows processes to catch and ignore segmentation fault signals; for some applications this is necessary semantics.) If a process is allowed to indefinitely ignore segmentation faults, this procedure will eventually find all segments in the address space. Security is compromised when it finds the first one; we analyze how long this will take.

Memory Size	1s Delay	1ms Delay	1μs Delay
`16MB`	24162 years	24 years	1.25 weeks
`256MB`	1510 years	1.5 years	13.2 hours
`2GB`	188 years	2.3 months	1.7 hours
`16GB`	23 years	8.5 days	12 minutes

Table 1: Time To Breach Anonymous Security Through Brute-Force Search

The analysis has the following parameters: V, the size of the virtual address space, M, the amount of physical memory [1], and D, the delay a process incurs on a segmentation fault. To simplify the formula, both V and M are in units of the maximum segment size. The result of the analysis is T, the expected time for a malicious process to find a segment for which it does not have the capability.

We first define $P(n)$, the probability that n memory probes will not find a mapped segment in n tries, or in other words, after nD seconds. $P(n)$ is equal to the probability that, in a sequence of V references, all of the references to the M mapped segments are after the first n references:

$$P(n) = \frac{\binom{V-n}{M}}{\binom{V}{M}}$$

To find T, we set $P(n) = 0.50$ and solve for nD. Table 1 presents some numerically calculated values for T, as a function of the delay and the amount of mapped memory, assuming a 64-bit address space and 8KByte segments. The segment size has little effect on the results presented in Table 1.

So the processor's natural fault-handling delay is unlikely to suffice, and the system must impose an additional delay penalty. The penalty should not only put the faulting process to sleep; to keep a malicious user from using multiple processes to search the address space, the penalty must also apply to all processes forked by the process's owner. However, this would still allow groups of users to divide the penalty.

The penalty time is best set as local policy. Constraints are the desired level of security, the number of users, and the added difficulty of working with faulty programs. For an currently average system, a constant delay of 1 second would seem acceptable on all fronts. As the memory size or user base of a system increases, it may be necessary to switch to adaptive delay functions which examine recent fault history. Fault history can also be examined to report suspicious patterns to the administrator.

Thus, the security we provide is not perfect; it is only probabilistic. However, we believe that the probabilities involved are low enough to make them of little concern when compared to external security issues.

Faulty software is also a threat; a faulty program may accidentally reference data it does not own. Normally, though, a failing program will stop once it causes a segmentation fault, and the likelihood that any individual accidental misreference will be to an otherwise unknown segment is quite small. The safety we provide is again probabilistic, not guaranteed, but the probabilities involved are minimal next to the chances of damage from external sources.

[1] Note that M is the amount of physical memory, not the amount of the virtual address space that is in use. When faulting in a page not in primary memory, the operating system can explicitly check the segment permissions at the time of the page fault.

3 Anonymous RPC

Running separate processes in the same protection domain will provide fast context switching; to take advantage of this, we must devise an interprocess communication mechanism that preserves anonymity.

Traditional UNIX paradigms like pipes and sockets are not easy to use for fine-grained communication; they involve considerable system and application overhead. A better scheme for our purposes is remote procedure call (RPC) [Birrell & Nelson 1984]. In RPC, a *server* process exports an interface to one of its procedures; any *client* process can then bind to the procedure as if it was linked directly into the client. Local RPC has been extensively studied [Bershad et al. 1990, Bershad et al. 1991, Schroeder & Burrows 1990], and seems to be the most convenient communication paradigm for integrating software systems across domains.

Druschel and Peterson optimized their RPC system mainly for data throughput; we feel that this goal has been achieved, and optimize our anonymous RPC system – 'ARPC' – for round-trip latency.

3.1 Maintaining Anonymity During Communication

In principle, there is nothing to stop two processes running in the same address space from communicating directly via procedure calls. Unfortunately, this cannot be used as a protected communication protocol. In a normal procedure call, the caller must know the procedure's address in the callee's code segment; and must tell the callee its own address, to allow return. This is incompatible with anonymity. Even if code segments are write-protected and all data segregated, a malicious process can trace through code to find data.

To preserve anonymity, the path of control must flow through some *intermediary*: an entity which is itself protected, aware of the RPC binding, and able to manage control and data flow without revealing either party's address to the other.

The most obvious intermediary is the kernel. Once a process traps to the kernel, it loses control of its execution, and can be shifted to any domain without having learned where that domain is. This is simple; unfortunately, it is slow. Kernel traps are typically an order of magnitude more expensive than procedure calls.

A better system can be devised if the host architecture supports execute-only page protection. Execute-only code lends itself well to anonymity; jumping to an execute-only entry point is anonymous for both sides. The caller knows the address of the callee's text, but cannot damage that text or discover where the data might be. We cannot give the caller the actual entry point in the callee's code, however; a jump into an arbitrary point in callee code might compromise data. Instead, we use an execute-only jump table, synthesized to contain the entry point as an immediate operand. The cost of anonymity is an extra jump. [2]

The data transfer protocol must also be modified to preserve anonymity. In a normal procedure call, the "server" uses the same stack as the "client" for local storage. It is therefore easy for either to corrupt the other by accidentally or maliciously writing into the wrong stack frame. Thus our RPC protocol must include a stack switch on call and on return. We must also clear registers which may contain data that can compromise anonymity.

Otherwise, anonymous RPC bears a close resemblance to ordinary procedure call, and can in some cases be performed with comparable efficiency.

3.2 ARPC Protocols

In this section we describe our ARPC protocols in considerable detail. A local RPC protocol can be constructed using either intermediary scheme. As an optimization, we design not one protocol but several, dividing RPC into several cases based on the level of *trust* between the client and server processes.

[2] Execute-only anonymity also requires an architecture on which branches are atomic; that is, they fully commit the processor to execution at the branch point. Some RISC architectures allow the processor to take a branch, execute an instruction, and then have the branch nullified by another branch in the delay slot.

If process A trusts that it is communicating with a non-malicious, but possibly buggy, process B, then process A can rely on the compiler and the RPC stub generator, instead of the kernel, to preserve its anonymity. This allows a more efficient implementation of cross-domain RPC. Malicious users can circumvent these utilities, but benign users are unlikely to do so accidentally.

3.2.1 Binding

Before any RPC calls can be performed, the client must *bind* to the server procedure. Binding is only performed once for each client, server, and procedure; thus it it need not be optimized as heavily as the call sequence itself.

The server initiates the binding sequence, by registering an *entry point*; it gives the RPC manager the name and address of the procedure. Once the entry point is present, the client can *connect*, telling the RPC manager the name of the procedure it wants to access. A permission check may be imposed on the connection; if it succeeds, the RPC manager generates the binding.

At bind time, the RPC manager creates any necessary intermediaries, and reports the RPC entry (be it an execute-only jump table, a kernel trap, or a direct entry into the server) to the client. The manager also creates an execution stack for use in executing calls through the new binding. All our ARPC protocols statically allocate one stack per procedure binding. This may seem wasteful of memory, but stacks can be cached and unmapped when not in frequent use. The static approach eliminates the need for a dynamic stack allocation on every call.

3.2.2 ARPC, Mutual-Distrust

Our first protocol is for the most general case, when neither client nor server trusts the other. In this case we cannot jump directly from client to server, even anonymously; we also have to save the client's stack and return address. This must be done in the intermediary. [3] The protocol is outlined in Figure 1.

```
push arguments and return address on client stack
enter intermediary
save return address
save registers
clear registers
save address of client stack
copy arguments to server stack
switch to server stack
leave intermediary to server procedure

execute server procedure

enter intermediary
copy return data to client stack
switch to client stack
restore saved registers
clear unsaved registers
leave intermediary to client return
```

Figure 1: ARPC Protocol, Mutual Distrust Between Client and Server

[3] If execute-only page protection is providing the anonymity, we jump to the intermediary through through an execute-only jump table.

For cross-domain procedure calls with few arguments, the dominant cost is saving, clearing, and restoring the registers. This is particularly true on modern processors with large register sets. For instance, on machines with register windows, the entire register set, not only the current window, must be saved, cleared, and restored on each call and return.

Marshaling of indirect parameters may not be necessary if the client organizes its memory properly. Since all addresses in the client are valid in the server, the client can keep data structures to be exported in a segment mapped separately from its private data, and pass pointers. Only if this is infeasible will explicit marshaling be needed. Direct parameters are marshaled on the client stack by the ordinary procedure-call protocol.

Note that in the absence of marshaling and the presence of execute-only code, no client stub is required. The client's view of the remote call can simply be a function pointer whose target address is the jump instruction in the anonymity table, and a generator-supplied header file can define its dereference as the function call. The ordinary argument-pushing semantics of a function call are exactly what we want.

3.2.3 ARPC, Server Not Malicious

The second protocol applies to a much more common case, when the server is trusted but the client is not. This might be the case, for example, in a small-kernel operating system. The protocol is in Figure 2.

```
save live registers
push arguments and return address on stack
call through intermediary to server

push address of client stack
switch to server stack
execute body of server procedure
pop address of client stack
push return data onto client stack
clear sensitive registers
return through intermediary to client procedure

restore live registers
```

Figure 2: ARPC Protocol, Server Not Malicious

Trusting the server not to be malicious has a considerable performance advantage. The client can anonymously jump (through the kernel or a jump table) directly to the server. In the simple case, the jump would be to a stub which would then marshal the client arguments onto the server stack. But we can achieve better performance by using an RPC generator to do a simple source code transformation of the server procedure so that it reads its arguments directly off the client stack; in this case, no copying is required. The RPC generator parses the server procedure, converts all argument references into client stack references, adds the instruction to save the client stack, and replaces ordinary returns with RPC returns.

This protocol is considerably more efficient than the mutual-distrust version. However, some additional work is required to make it protocol safe and secure.

Safety could be compromised if a register containing a client pointer became the initial value of a server variable. The solution is to ensure that all automatic server variables are initialized before use. A program which relies on a the value of an uninitialized variable is unlikely to be correct, and most modern compilers can at least warn of such errors.

Likewise, security could be compromised by server data passing through registers back to the client. This is a more serious problem. The server must clear all sensitive registers before returning; if compiler analysis cannot identify which registers are sensitive, all registers must be cleared.

Another safety hole is the client stack. Although we allow the server to access its arguments on the client stack (to eliminate copying), we do not want the server to be able to inadvertently 'smash' the client stack. The solution is to prevent the server procedure from taking the address of any of its arguments; this can be enforced in the RPC generator.

3.2.4 ARPC, Neither Side Malicious

Finally, our third case: a protocol for cases in which both client and server are trusted to be non-malicious, in Figure 3.

```
save live registers
push arguments and return address on stack
call directly to server procedure

push address of client stack
switch to server stack
execute body of server procedure
pop address of client stack
push return data onto client stack
return directly to client procedure

restore live registers
```

Figure 3: ARPC Protocol, Neither Side Malicious

The only operation being performed here which is not part of a normal procedure call (assuming the caller-save register protocol) is the stack switch. And the protocol is safe; data may cross in registers, but a correctly-compiled program will never allow it to be addressed by a variable.

3.2.5 Service Management

Like LRPC [Bershad et al. 1990], ARPC is implemented by running the server procedure in the client thread. Any alternative would involve a slow interaction with operating system data structures. The logical semantics of RPC, however, imply a sleeping service thread which awakes to run the procedure and returns to sleep when it finishes. Reconciling these models requires some juggling.

One problem comes when the server crashes. We do not want the client thread to die with the server, because it was not the client who caused the error. Instead, the RPC call should return with an error condition.

Our solution is expensive, but not inappropriate given that such faults should be a rare condition. We check all the server's incoming intermediaries for saved stacks, and restore the client threads from those; if the server has any outgoing calls, we zero the stack in the intermediary so that the call faults when it returns.

This does not work for the mutual-trust protocol, which has no intermediary and saves the client stack address on the server stack. If separate crash recovery is required, a separate word must be reserved for the client stack. The separate-recovery model, in any case, is often not the preferred semantics for mutual-trust uses.

Synchronizing stack allocation is another problem. All of these routines assume one proviso: that only one thread of control, including past levels of recursion, is using the same RPC binding at any time. The assumption is convenient because it allows stacks to be statically allocated. If multiple threads must use the same RPC binding, however, they will have to synchronize externally, as they would for any single-consumer resource.

Also, in systems which support threads, a simple lock is necessary at the beginning of untrusted-client entry points, to prevent malicious clients from sending multiple threads through the same binding and causing a stack collision.

4 Uses of Anonymous RPC

ARPC can perform many functions within a system. It can be installed at the user level, to provide additional safety within large applications that are externally protected by traditional protection domains; or it can be installed at the system level, as the main interprocess protection system; or it can be used only for some specialized tasks which require especially low latency. We consider each.

4.1 User-Level ARPC

Many large programs are written in languages, such as C, which do not guarantee the internal safety of memory. That is, code in one module may inadvertently corrupt the unrelated data of another, creating a bug which is hard to find in single-threaded code, and almost intractable in a multithreaded program.

This is an unpleasant possibility, but software designers accept it because of the high performance penalty a safe implementation would incur. In conventional systems, safety can only be ensured by actively checking each pointer dereference, or by placing separate modules in separate hardware protection domains. Either solution incurs a substantial performance penalty and neither is widely used.

ARPC offers a convenient medium. When neither side is malicious, the cost of an ARPC call is little greater than the cost of an ordinary procedure call. In most cases, replacing procedure calls with ARPC calls will have little impact on performance.

We can use this principle to increase internal safety in large software systems. Logically separate modules, and their heaps, can be placed in separate ARPC domains, providing strong probabilistic safety at minimal cost.

Such a transformation can be installed transparently in a C compiler and linker, providing increased safety invisibly to the programmer. It can also be used in compilers for languages that do guarantee memory safety, allowing them to maintain their safe semantics while providing performance competitive with C. Protecting small objects can waste memory: an anonymous object smaller than a page must still take take up an entire page of physical memory.

4.2 ARPC as an Operating System Base

With some care, ARPC can be used as the sole process protection system in a traditional UNIX-style operating system. This would accelerate communication between user-level processes; it might, for example, allow fast, fine-grained drawing interaction between imaging software and a display manager.

It can also be used, in a more radical design, to improve the internal structure of the operating system. ARPC allows considerable kernel decomposition, even beyond the usual microkernel level of [Young et al. 1987]; all traditional system services – filesystem, communication, scheduling, memory management, and device drivers – can be performed at user level.

This is possible because, in an ARPC system, all common operating systems primitives can be executed without supervisor privilege. Context switches do not require manipulation of the virtual memory hardware, and process authentication can be performed by giving system servers separate ARPC entry points for each process. Even page tables and other hardware-accessed data structures can be mapped into the user address space and used directly by user-level servers. Memory-mapped devices can be treated the same way. The only functions that must be performed at supervisor level are system initialization and execution of privileged instructions. A very small kernel can handle the latter, verifying that requests come from genuine system servers by checking the source address of the trap.

This model has some practical deficiencies. UNIX semantics require separate address spaces for sibling processes after a fork(); although the extra space can be discarded upon the first exec(), handling it may prove a considerable design nuisance.

We also caution against using ARPC as the sole protection device in applications where local security is mission-critical. In an ARPC-based system, it seems too easy for small software errors in implementation – the accidental release of capabilities – to leave obscure security holes that could be exploited by dedicated intruders. This is true in any system that uses cryptographic capabilities – e.g. Amoeba [Mullender et al. 1990] – but especially so in ARPC, where every pointer is a capability.

4.3 Limited Uses of ARPC

To be useful, ARPC need not be the main protection device in a system. Its use can be restricted to special circumstances where speed is important, with traditional hardware domains used in all other places where protection is needed.

One such use might be network communications. Some new networks [Anderson et al. 1992] offer latency on the order of microseconds, a useful feature which traditional software architectures have difficulty exporting to the user. An ARPC solution would be to map the network device anonymously and give its address to a user-level device manager, which would in turn accept ARPC calls. Network-critical processes would have to share an address space with the device manager and the device; all other tasks could have their own spaces.

5 Some Problems and Disadvantages of ARPC

One potential problem with an ARPC-based environment is that image initialization is slower; since programs must be remapped randomly every time they are instantiated, they must be relocated on execution. This is a concern, but the latency of initialization is already great and relocation can be performed as part of copying the program into memory.

An anonymous address space may be very sparse and difficult to manage. If logical domains become as small as one page, a traditional multilevel page-table scheme will be prohibitively expensive. However, inverted page tables [IBM 1990] will work; as will a software-loaded TLB [Kane 1987] backed by a simple search scheme such as binary tree or hash table.

Tagging TLBs and virtual caches with process identifiers would decrease the cost of context switches and makes conventional protection more competitive with ARPC. Tagged TLBs and caches eliminate the need to flush state on context switch, one of the most expensive components of a traditional domain switch. However, not many architectures support tagged TLB's; even if they do, the switch must still be performed in supervisor mode.

It is also uncertain whether the imbalance of virtual and physical memory will continue. One cannot predict whether or not the industry will move to a 128-bit bus before available physical memory approaches 64 bits. It is worth noting, however, than in the past bus size has increased faster than physical memory size.

Also, ARPC precludes other uses for the 64-bit address space which may be superior. Large databases [Stonebraker & Dozier 1991] or distributed systems [Carter et al. 1992] can use most of a 64-bit space.

6 Performance Results

We implemented a test prototype of anonymity on an Intel 486-33 machine, capable of about 15 SPECint. The base operating system was Linux 0.98.4, a copylefted POSIX clone for the 386 architecture [Torvalds 1992]. The 486 is a 32-bit machine, and as such a truly functional implementation was impossible; however, we did our best to assure that practical concerns were treated as realistically as possible.

We did not convert the entire system to use a shared virtual address space; only specially-flagged relocatable executables were run in the same address space. Even on the 32-bit 486, it would have been

Protocol	Time (μs)	Processor	MIPS	μs/MIP
`SRC RPC`	454	C-VAX	2.7	1226
`Mach RPC`	95	R2000	10	950
`LRPC`	157	C-VAX	2.7	424
`URPC`	93	C-VAX	2.7	251
`ARPC`	7.7	i486-33	15	116

Table 2: Null RPC Performance Results

feasible to run all processes in the same space, but Linux is distributed largely in binary form, and finding and rebuilding source for all our utilities to make them relocatable would have been a task not worth the benefit.

Under this system we chose to test the slowest possible version of RPC; the protocol for two mutually untrusted processes, implemented without the use of execute-only code. Although the intermediary was accessed by traps, it ran in user mode.

Our time for a user-to-user round-trip null RPC in C was 7.7 microseconds. Of this, 3.4μs were due to the user-level traps; 2.4μs to segment peculiarities associated with the Linux implementation; 0.5μs to save, clear, and restore registers; and 1.8μ for overhead in the intermediate region. That overhead was 29 instructions; of them, 15 were involved in loading static parameters, and could be eliminated were we to synthesize the intermediary for each bind [Massalin & Pu 1989]. This is somewhat more overhead than we had hoped for, but we consider it acceptable.

It is difficult to find comparable performance figures for other RPC systems, since we do not know of any other optimized local RPC results on the same architecture. Instead, Table 2 compares our performance to that of four other RPC implementations running on other hardware: Mach RPC [Bershad et al. 1992], SRC RPC [Schroeder & Burrows 1990], LRPC [Bershad et al. 1990], and URPC [Bershad et al. 1991]. These are all optimized RPC implementations; commercial RPC implementations are frequently another order of magnitude slower.

7 Related Work

The most closely related work to ours is Druschel and Peterson's *fbufs*, packet buffers mapped in a shared anonymous space [Druschel & Peterson 1992]. User-level protocols communicate via fbufs to exchange packets at very high speed. Fbufs are a much more robust and conservative use of anonymity; the user-level protocol layers which use fbufs to communicate each have their own protection domain. Performance improvements are achieved by increased flexibility and efficiency in buffer management. Buffers do not need to be remapped in the middle of a transfer, nor must they be assigned to specific protocol pairs beforehand. Fbufs achieve data transfer rates near the theoretical limits of memory.

Other systems, like Pilot [Redell et al. 1980], use a single address space without separate hardware domains, and rely for protection on languages which restrict the use of pointers. Pilot was designed for installations which do not need security, like some personal computers, and thus benefits from the ability to optimize its protection mechanisms only for safety.

Wahbe et al. [1993] propose a different approach to enforcing protection. Instead of relying on hardware, or on anonymity, they enforce protection in software by installing extra instructions into the object code to prevent out-of-bounds pointer dereferences. Unlike Pilot, Wahbe et al.'s approach is language-independent because the protection is enforced at runtime, not in the compiler. The benefit is that separate processes can run in the same address space and communicate efficiently, at the expense of a slight slowdown in memory operations.

The Opal system [Chase et al. 1992] also uses a 64-bit shared address space, but each Opal task has its own protection domain; the goal is uniformity of naming. Using identical addresses to reference the same objects in all domains allows increased flexibility in sharing complex data structures. A context switch, however, still involves switching page tables. Other systems also follow this pattern [Carter et al. 1992].

8 Conclusion

Anonymous RPC exploits a simple property of memory systems: that it is impossible to address data whose address is unknown. With the advent of 64-bit machines, it is now possible to take advantage of this property, by placing segments at random locations in a very large, sparse address space. This allows efficient interprocess communication without expensive hardware domains.

We believe that ARPC is a useful technique for high-speed communication between closely coupled domains. Even if it is not used to organize an entire operating system, it can be used as to protect individual subsystems which use especially fine-grained communication. We suggest that software designers consider it as an option, and that hardware designers consider including features, such as execute-only page protection, which help its implementation.

9 Acknowledgements

We would like to thank Peter Druschel, Dave Keppel, and Ed Lazowska for their helpful comments on this paper.

References

[Anderson et al. 1991] Anderson, T., Levy, H., Bershad, B., and Lazowska, E. The Interaction of Architecture and Operating System Design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 108–120, April 1991.

[Anderson et al. 1992] Anderson, T., Owicki, S., Saxe, J., and Thacker, C. High Speed Switch Scheduling for Local Area Networks. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 98–110, October 1992.

[Bershad et al. 1990] Bershad, B., Anderson, T., Lazowska, E., and Levy, H. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1), February 1990.

[Bershad et al. 1991] Bershad, B., Anderson, T., Lazowska, E., and Levy, H. User-Level Interprocess Communication for Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2), May 1991.

[Bershad et al. 1992] Bershad, B., Draves, R., and Forin, A. Using Microbenchmarks to Evaluate System Performance. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992.

[Birrell & Nelson 1984] Birrell, A. and Nelson, B. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Carter et al. 1992] Carter, J. B., Cox, A. L., Johnson, D. B., and Zwaenepoel, W. Distributed Operating Systems Based on a Protected Global Address Space. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992.

[Chase et al. 1992] Chase, J., Levy, H., Baker-Harvey, M., and Lazowska, E. Opal: A Single Address Space System for 64-bit Architectures. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992.

[Dennis & Van Horn 1966] Dennis, J. B. and Van Horn, E. C. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[Druschel & Peterson 1992] Druschel, P. and Peterson, L. L. High Performance Cross-Domain Data Transfer. Technical report, Department of Computer Science, University of Arizona, 1992. Technical Report 92-11.

[IBM 1990] IBM Corporation. *POWER Processor Architecture*, 1990.

[Kane 1987] Kane, G. *MIPS R2000 RISC Architecture*. Prentice Hall, 1987.

[Massalin & Pu 1989] Massalin, H. and Pu, C. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 191–201, December 1989.

[Mullender et al. 1990] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer Magazine*, 23(5):44–54, May 1990.

[Nat 1977] National Bureau of Standards. *The Data Encryption System*, 1977. Federal Information Processing Standards Publication 46.

[Redell et al. 1980] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[Schroeder & Burrows 1990] Schroeder, M. and Burrows, M. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[Stonebraker & Dozier 1991] Stonebraker, M. and Dozier, J. Sequoia 2000: Large Capacity Object Servers to Support Global Change Research. Technical report, Computer Science Division, University of California, Berkeley, July 1991.

[Torvalds 1992] Torvalds, L. *Free Unix for the 386*, 1992. finger torvalds@kruuna.helsinki.fi.

[Wahbe et al. 1993] Wahbe, R., Lucco, S., Anderson, T., and Graham, S. Low Latency RPC Via Software-Enforced Protection Domains. Technical report, Computer Science Division, University of California, Berkeley, April 1993.

[Young et al. 1987] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 63–76, November 1987.

10 Author Information

Curtis Yarvin is a first-year graduate student in the Computer Science Division at the University of California at Berkeley. He graduated from Brown University in 1992. His e-mail address is "curtis@cs.berkeley.edu".

Richard Bukowski is a first-year graduate student in Computer Science Division at the University of California at Berkeley. He graduated from Cornell University in 1992. His e-mail address is "bukowski@cs.berkeley.edu".

Thomas Anderson is an Assistant Professor in the Computer Science Division at the University of California at Berkeley. He received his A.B. in philosophy from Harvard University in 1983 and his M.S. and Ph.D. in computer science from the University of Washington in 1989 and 1991, respectively. He won an NSF Young Investigator Award in 1992, and he co-authored award papers at the 1989 SIGMETRICS Conference, the 1989 and 1991 Symposia on Operating Systems Principles, the 1992 ASPLOS Conference, and the 1993 Winter USENIX Conference. His interests include operating systems, computer architecture, multiprocessors, high speed networks, massive storage systems, and computer science education. His e-mail address is "tea@cs.berkeley.edu".