

Dynamic Service Reconfiguration and Migration in the Kea Kernel

Alistair C. Veitch and Norman C. Hutchinson

Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada V6T 1Z4
{aveitch,norm}@cs.ubc.ca

Abstract

Kea is a new operating system developed for experimentation with kernel structuring, configuration and specialization. There are several features of Kea's design that make the investigation of these issues practical. Firstly it supports fine-grain decomposition of kernel services, the components of which communicate using inter-domain calls. This communication mechanism forms the backbone of Kea's reconfigurability, as services can be located in separate domains, for development or debugging purposes, and then dynamically migrated to a common domain, or into the kernel itself, transparently to the users of the service. The inter-domain calls are automatically optimized to procedure calls as appropriate. The service hierarchy can also be dynamically reconfigured through replacement, or the layering of new services, either on a system wide or application specific basis. We describe these features, and discuss the results from several experiments that demonstrate the practicality and performance advantages of Kea's design.

1 Introduction

Kea is a new operating system kernel developed for experimentation in kernel structuring, configuration and specialization. There are several reasons why such capabilities are becoming increasingly important. Firstly, the issue of how best to design, configure and build an operating system kernel continues to be an issue in the research community. While many paradigms are possible, the primary two are the microkernel and monolithic designs. Monolithic kernels, in many ways the "traditional" design, place all the system functionality in a single image, usually compiled from many hundreds of thousands of lines of code. Microkernels aspire to have the kernel be as simple as possible, providing only the minimum abstractions required to represent the underlying hardware, and letting the rest of the system, such as network protocols, file systems, user processes etc., be provided by a collection of user level servers. These servers have to communicate with the kernel using some form of inter-process communication (IPC), often with remote procedure call (RPC) layered on top. The microkernel approach encourages a separation of concerns, and allows for the easier development of services, as they can be written as

user level tasks which are consequently easier to debug. Additionally, a fault in one server can be restricted to a single address space. However, due to the communications overhead imposed by the RPC layer, microkernels cannot be as fast as monolithic kernels. In addition, microkernels almost always implement the other operating system services in a single server, which contains almost identical code to a monolithic system, negating many of the modularity, maintainability and structuring possibilities inherent in the microkernel design.

The design of the Kea system provides a solution to the performance problems inherent with microkernels, while retaining, and in fact encouraging, the modularity and incremental development advantages of the microkernel philosophy. Kea achieves this by providing the means by which applications, and the kernel itself, can be dynamically reconfigured. The central paradigm which Kea uses to achieve flexibility is the *service*. A service has two parts, an interface and an implementation. The interface describes the procedures, constants and data types that the service offers to clients, while the implementation contains the compiled code. Services can be dynamically replaced, created and interposed in an incremental and safe manner, either on a global or application specific basis. This allows the system to be reconfigured or specialized in many ways, depending on the granularity of services, and the relationships between them. In particular, services may be dynamically migrated into the kernel, removing the RPC overhead.

This paper examines the Kea primitives that facilitate these abilities, and describes the results from several experiments to measure and evaluate them, notably in the area of file system reconfiguration and layering. Section 2 introduces the Kea architecture, describing the basic features of the system. Sections 3 and 4 describe the two most important primitives, service migration and service reconfiguration respectively, with brief details on their implementations. Section 5 describes some of the experiments we have performed. Section 6 compares Kea to several other systems with similar goals or techniques. Finally, the paper is summarized in section 7.

2 Kea Architecture

This section gives a brief introduction to many of the main features of the Kea system. Several of these are described in

greater depth in a previous paper [20].

Kea is essentially a microkernel, in that it is constructed as a small kernel (approximately 90 Kb in the current version) which provides abstractions of a machine's physical resources (memory, CPU, interrupts etc.) which can then be used to construct higher level services. In a traditional microkernel, the various services communicate using some form of IPC, often with some form of RPC layered on top. While Kea uses a form of RPC for communications between services, this is also where Kea departs from traditional microkernels. The RPC mechanism, which we refer to as an *inter-domain call* (IDC) in Kea, is changeable in several different ways, making the reconfiguration of services possible. Before discussing exactly why IDC is different, and how it is used for flexibility, we need to introduce several more concepts, notably *domains*, *threads*, and *portals*.

2.1 Domains, Threads, and IDC's

A domain is essentially a virtual address space which supports standard operations such as memory allocation, page sharing and copy-on-write. Domains are also the system entities which own other resources, and thus form the basis for system security.

Threads are the units of execution, and as in several other systems [8, 10], are split into two parts, a thread body and a stack of *activations*. The body holds global thread information, such as the thread identifier, kernel stack, scheduling parameters and security information. Each activation contains storage for a machine's register set, and a collection of per-domain information, including a stack. By pushing or popping activations onto or off of a shuttle's activation stack, a thread's execution context can move between address spaces. It is this act of a thread moving between domains which comprises an IDC. As the thread/IDC mechanism allows for the direct migration of threads between protection domains, it is inherently more efficient than the IPC and RPC layers offered in a conventional microkernel. There are several factors contributing to this efficiency. Firstly, there is only one layer of processing, not two (i.e. IPC with RPC layered on top). Secondly, since any parameters to be passed between activations have the same format on both caller and callee, it is possible to use the machine's native data encoding, rather than a canonical form. Finally, since the same thread is running throughout the call, the scheduler does not have to be involved, effectively removing a level of thread synchronization. As a further optimization, Kea also keeps caches of activations and stacks.

2.2 Portals

Portals are the abstraction that mediates the use of IDC. A

portal contains information about a destination domain and an entry point in that domain. Through the system call `portal_invoke()`¹, a thread requests to the system that an IDC be made to the domain and entry point represented by the portal. Once a domain has acquired a portal, it can use it to obtain access to the service backing that portal, without the programmer having to know which domain the portal is located in. A domain sees a portal as an integer; the globally unique system identifier for the system maintained per-portal information.

The IDC/portal system makes an IDC appear to be an ordinary procedure call. Each portal contains explicit information on the expected arguments for each call. At the time the IDC is made, the kernel copies the arguments to the remote domain. Pointer arguments are transferred by allocating space in the remote domain, copying the data, and passing the new pointer value.

2.3 Services

While portals provide a means by which individual calls may be made between domains, they do not, by themselves, offer much in the way of a structuring mechanism. For this, Kea uses what we call services, which are also the central paradigm which Kea uses to achieve flexibility. A service is essentially a well defined interface through which some entity can be accessed and asked to perform a task for the caller.

There are several ways to view a service. As seen by the programmer, there are two basic parts to each service, an interface and an implementation. The interface describes the procedures, constants and data types that the service offers to clients, while the implementation contains the compiled code. A reasonable comparison to make is that the interface is analogous to a C header (“*.h*”) file, while the implementation is like a library. As an example, the set of procedures that manipulate the virtual memory of a domain are grouped into a single service, the “*vm*” service. The service guarantees to provide the given procedures, with specified semantics. From the application's viewpoint, a service appears as a collection of function pointers – one for each of the service entry points. By dereferencing these pointers and calling the specified function, the service can be accessed. To both the programmer and application, this appears as a local function call². The reason for using function pointers as the entry point for services will become clear in the later sections on portal migration and remapping. The functions pointed to by these pointers are automatically generated stubs, which do little more than call `portal_invoke()` with the correct portal identifier. These stubs are linked with the service when it is

1. `portal_invoke()` is the only system call in Kea.

2. This is at least true for the C language, in which Kea has been developed.

loaded. From the kernel's viewpoint, a service is essentially a set of portals. These portals are manipulated as a group, ensuring the consistency of the service – the programmer never has to be concerned about the underlying portal representation.

2.4 Service Naming

Services are referenced in the system by name. For instance, the service implementing the keyboard driver is known as “keyboard”, while the IDE disk driver is known as “ide”. However, many services may need to be duplicated, where each duplicate provides an almost identical service, but differs in some small but important way. An example of this is a filesystem, where there may be multiple filesystems of the same type (e.g. a DOS FAT filesystem, or BSD FFS) on the machine's disks. In this case, there needs to be some way of separating them. Kea accomplishes this by separating services, which specify the high level details such as number of entry points and the details of the arguments to be used for each procedure, and *instances* of services, which specify the domain in which a particular instance actually exists and the entry points within that domain. Thus, there might be one DOS filesystem service, called “dos”, but two instances of that service, one called “ide0/dos0” and another “ide0/dos1”, respectively representing the first and second DOS filesystems on the first IDE disk. For simplicity, we will ignore the existence of instances for the remainder of this paper, and refer to services as if they were a single entity.

2.5 Portal Remapping

Services are the central paradigm which Kea uses to achieve flexibility and reconfigurability. Before discussing how this is accomplished, it is important to examine some of the operations that can be performed on the underlying portals, as they determine the scope of the service reconfiguration.

We refer to the portal reconfiguration operations as *remappings*, as they cause a portal to have the destination (server) side remapped to a different endpoint. There are two primary types of remapping. The first of these, which we refer to as *simple* portal remapping, is simply a change to either, or both of, the domain and entry point. This change is transparent to the application, i.e it continues to use the same portal.

The second, and more complex, form of remapping is shown in Figure 1. Here, two applications, A and B, are both using a portal to access a service, S_1 , which in turn uses a service S_2 . It is possible for B to specify that, whenever S_1 is using S_2 on it's behalf, it wants an alternative service S_3 to be used instead. In this case, S_1 continues to use the same portal to obtain a service, but because S_1 is performing the service on behalf of B, the destination of the IDC is not S_2 ,

but S_3 . This is known as domain-rooted remapping, and can be used to support several different varieties of application specific configuration.

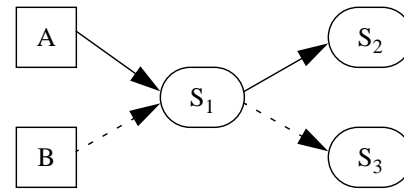


Figure 1. Domain-rooted Portal Remapping
Solid lines show IDCs for A, dotted lines for B.

3 Service Migration

There are several ways in which services can be manipulated. The first of these is through migration. When a service is first loaded, the system allocates space for the service's text and data in a domain, creates the portals needed for access to that service, and calls the service's initialization function. The service can be located in either a newly created, or an already existing, domain. As the system is used, it may be desirable to move the service into another domain. The primary reason to do this is performance. If a service can be migrated into the domain of the most frequent client of the service, then the time needed for each IDC can be reduced by the time taken for changing address spaces, which is a potentially expensive operation. Alternatively, IDCs into the kernel are far cheaper than those into another domain. Thus, for trusted services such as device drivers, filesystems, and network protocols, it is desirable to move these into the kernel once they have been debugged. This offers several advantages to the service developer, particularly if they are developing kernel services. Firstly, the strictly defined service interfaces enforce modularity, which makes the system as a whole more maintainable. Secondly, the kernel developer sees exactly the same programming interface as the application developer. The kernel makes all its low level interfaces available as services. This unification of the user and kernel programming interfaces makes it easier for programmers to develop kernel code. Thirdly, it is not uncommon for newly developed kernel services to cause a system crash, necessitating a tedious program/crash/reboot development cycle. Within Kea, services can initially be developed within a private domain, which restricts the crash (if any) or consequences of a programming error, to that domain. Additionally, since the service is running in a separate domain, it can be more easily debugged with standard tools.

When compiled, services are only compiled as far as single object files. When loaded into a new domain, this

object file is dynamically linked against any necessary libraries. When the service is loaded into, or migrated to, an already extant domain, it is first dynamically linked against the existing application and/or services in that domain. This prevents any duplication of routines within domains, while keeping service code separate. To accomplish this, the kernel must keep symbol information for each domain and service available, but we have found this to only be a minor cost – for example the storage space needed for the symbols in our standard C library is only 5 Kb. The current service migration mechanism migrates only the text and statically allocated data between services. Any state inherent in the service (e.g. active file descriptors in a file system) must be packaged up by the service into a given memory area, which is provided to the kernel as part of the migration. When the service is restarted in its destination domain, a pointer to this data is presented to it as an argument to the service initialization function, allowing the service to recover the state. We believe that this state could be automatically transferred using new techniques for process migration [18], but are reserving this possibility for future work.

As a performance optimization, the kernel detects when services are colocated, i.e. when two services are located in the same domain, and optimizes the IDC into a single indirect procedure call, eliminating the kernel trap overhead completely. This is easily accomplished as the access points to each service are kept in function pointers. Because the kernel has access to the symbol table describing these pointers, and knows the names of the relative routines, it can adjust the pointers to point to the service directly, rather than to the IDC stub. This eliminates one of the standard complaints about microkernel systems, namely that the performance overhead imposed by IPC/RPC is too costly. By giving control of service location to the system developer, and letting services be colocated when safe and appropriate, the Kea system can give the best of all worlds – safety, modularity and performance, depending upon the system requirements.

4 Service Reconfiguration

The second major type of per-service operation that can be dynamically carried out in Kea is service restructuring, either on a global or domain specific basis. By using portal remapping, services can be dynamically inserted or replaced at any point in a service hierarchy. This allows Kea to support several different styles of reconfiguration, including stackable systems [15, 3], interposition agents [9], continuous operation [13] and various other styles of application specificity [20].

Figure 2 shows just two of the possible configuration changes that can be made. In this figure, various service configurations are shown, proceeding from left to right. The

initial configuration, (a), shows two applications (A and B), using a chain of services S_1 through S_3 . In (b), A new service, S_4 , has been interposed between the applications and S_1 . S_4 offers the same semantics as S_1 , and the applications will not be able to tell that the underlying service structure has changed. This type of reconfiguration can be used to support several application needs. For instance, if S_1 was a filesystem, S_4 could offer compression services, transparently uncompressing and compressing the data for read and write calls respectively. Another example could be adding an encryption layer onto a standard network protocol. Adding a service does not necessarily have to happen at the top of the call chain, but can be performed anywhere in the service graph, for example between S_1 and S_2 . In all cases, service interposition can be carried out transparently to both the clients and users of all services – they continue to access, and be accessed, through the same portals as before the insertion.

Part (c) of Figure 2 shows an application specific remapping. Here a service S_5 has been inserted, taking the place of S_2 for all calls from application B. Service usage that originates in application B (shown by dotted lines) is redirected through S_5 . This effectively allows applications to make various adjustments, notably in policy decisions, that affect the application’s performance. There are several areas in which such application specific service remapping could be effective, including virtual memory management [14], file system cache management, or specialized network protocols [5, 6].

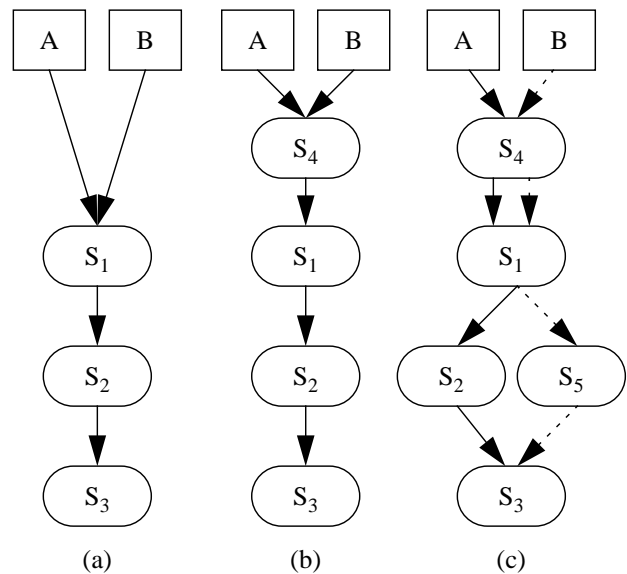


Figure 2. Service Reconfigurations

5 Experimental Results

We have performed several experiments designed to test

both the kernel's functionality and performance, and to show that the Kea design is applicable for systems requiring dynamic reconfiguration. We have focused these experiments largely on the file system. The primary components used in these experiments were an IDE disk driver, DOS FAT¹ file systems and a "global" file service that ties together other file systems in the system into a global namespace, effectively multiplexing among the applications using the file system and the per disk partition file systems. The general relationships between these services are shown in Figure 3. For experimenting with service interposition, we also implemented a compressed file service, that transparently compresses the data in read and write calls.

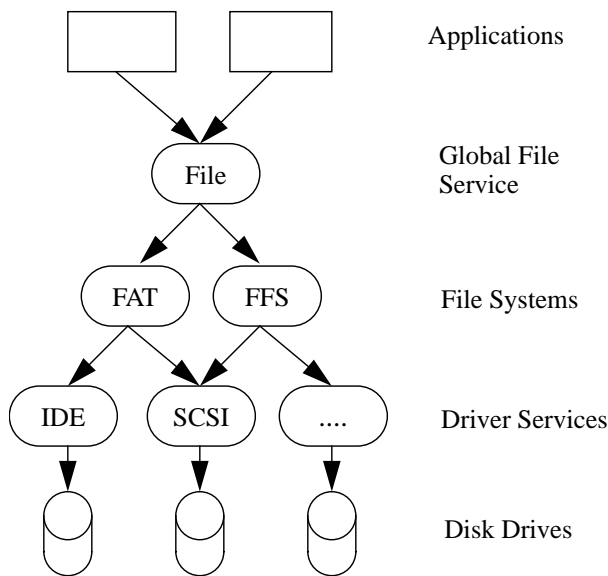


Figure 3. Experimental Services

All experiments were performed on a 100 MHz Pentium, with 256 Kb L2 cache, 64 Mb of main memory, and an 814 Mb IDE disk. The times were measured using the Pentium "rdtsc" (read timestamp counter) instruction, which returns the number of clock cycles executed by the processor since the last reset. On a 100 MHz machine, this lets us obtain timings with a granularity of 10 nanoseconds. For comparisons, we measured equivalent operations (where they existed), on the same machine running FreeBSD 2.2.2-RELEASE, in single user mode.

The first set of experiments were designed to verify that each of the services, developed at user level, was transferable to the kernel, and to evaluate the performance

1. As Kea is primarily developed on an IBM AT architecture, these were some of the first services developed. We have other services (such as a SCSI disk driver and preliminary BSD filesystem) and are working on others, but believe that these are sufficient for our initial experiments.

benefits accruing from this action. Co-locating the file systems and device drivers in the kernel is normal for most operating systems, as these are trusted services. Kea allows the development of these services independent of their final destination domain, facilitating safer development and providing some reassurances about the service behaviour. The first (bold) set of numbers in Table 1 shows the times taken for various file system operations, for various placement permutations (kernel or user level) of the services. All the times shown are in microseconds, We performed each operation twice, once as soon as the services started ("cold" times), and then again, to observe any changes due to "warm" caches and file system buffers. The first column shows times with all services in separate domains, the second with the IDE driver in the kernel, the third with the IDE and FAT services in the kernel, and the fourth with all services migrated into the kernel. The fifth column has results for all services co-located into a single domain (roughly equivalent to the usual microkernel "single server"), while the final column has results from FreeBSD for comparison.

Looking only at the results for Kea, it is obvious that for most operations, there is a substantial time saving involved in moving services into the kernel. Co-locating services in a user-level domain also shows a time improvement, although this is not as substantial as placement in the kernel. The biggest exception to this rule are the times taken for the "close" operation. This consumes substantially more time when executed with the FAT filesystem in the kernel. The explanation behind this is interesting, and illustrates an interesting trade-off between kernel and user level service location. When a file is opened, the filesystem allocates some space for buffers describing the file's layout on the disk. In the experiments described, the file used was 10 Mb, necessitating a relatively large amount of such buffer space. When closed, this buffer space is freed. The kernel memory allocator aggressively reclaims physical pages, and removes their virtual to physical mappings. The user level memory manager however, is lazier in its page deallocation, deferring some of the cost in the hope of reusing the pages at a later time. This results in apparent time savings for freeing larger amounts of memory from user level. We verified this through fine-grain measurements of the various operations in the "close" path. For further confirmation, and to provide more numbers for comparison, we also repeated the experiment for a much smaller (2 Kb) file. These results are the second set of numbers in the table.

As expected, almost all of the times are reduced, indicating that for smaller files, less disk I/O and buffer allocations are needed. The biggest exceptions are the FreeBSD "cold" open time, which increases by over 50%, and the read time, which only drops by 13%, as opposed to an average drop of 52% for Kea. We suspect that this is due

	Operation/ Locations	Separate domains	IDE in kernel	IDE/FAT in kernel	All in kernel	Single domain	FreeBSD
“cold”	Open	559 / 249	559 / 243	442 / 257	374 / 167	557 / 236	18614 / 28444
	Read 1K	19917 / 9482	19607 / 9335	19307 / 8555	19006 / 9007	19195 / 9559	20163 / 17463
	Write 1K	176 / 201	176 / 192	125 / 161	78 / 94	152 / 186	109 / 106
	Close	3258 / 3591	3392 / 3258	4589 / 3231	4489 / 3413	3112 / 3113	58 / 59
“warm”	Open	447 / 253	447 / 250	427 / 221	360 / 122	443 / 229	134 / 131
	Read 1K	184 / 199	184 / 187	155 / 139	85 / 72	123 / 188	95 / 90
	Write 1K	170 / 184	170 / 175	139 / 126	73 / 72	158 / 174	111 / 105
	Close	10198 / 9897	9843 / 10512	10389 / 10770	10599 / 10958	10122 / 10791	61 / 52

Table 1. File system operation times (μ s), 10 Mb / 2 Kb file

to inefficiencies in the FreeBSD I/O subsystem, but have yet to instrument their code sufficiently to trace this.

Comparing the results for FreeBSD and Kea is also interesting. For most operations, Kea is slightly faster, giving us confidence in our view that the fine grained, modular, construction of Kea’s services does not impose an undue performance penalty. The big exceptions to this are the “cold” open time, which is much slower for FreeBSD, and the close times, which are much slower in Kea. The first exception is, we believe, caused by FreeBSD having to read the FAT filesystems root directory structure on the first open, while Kea’s FAT filesystem caches this data on start-up. The close times are more problematic. Kea initiates disk I/O for any buffered writes to the file when the file is closed, and the close times include the time taken for these operations. FreeBSD doesn’t do this however, only writing file buffer caches periodically, rather than on close, effectively deferring the cost of the disk I/O to a later time. Also, the difference in Kea close times for the cold and warm cases are due to the tests timing differences resulting in a different disk head location, and hence longer rotational latencies. For these reasons, we do not believe that the close times are directly comparable. In fact, the general comparison of the times for the two systems should be looked at carefully, as they use different buffering and I/O mechanisms. In order to try and minimise the impact of this, we measured the time to perform a read or write of 1 Mb of contiguous file data, using 1 Kb data blocks. The results from this experiment are shown in Table 2. For Kea, all services were co-located in the kernel. This table shows that Kea is slightly slower for reading, but substantially faster for writing, a result that we find very favourable.

OS/Operation	Read	Write
Kea	0.683	2.02
FreeBSD	0.654	3.48

Table 2. Time to Read/Write 1Mb (seconds)

The second set of experiments to be performed involved testing the time taken to migrate each of the services from user to kernel space. This time is important to show that migrations can be accomplished in a reasonable amount of time. It also indicates the practicality of dynamic kernel upgrades, as the time taken for service replacement is the same as that for migration. The time taken to migrate a service depends on several factors – the amount of executable code comprising the service, the time needed to link this code into the new domain, the amount of service state to be transferred, and the time taken by the services initialization function to execute. To assess the times attributable to these factors, we made several measurements, which are shown in Table 3. The first column shows the service name, the second the size of the service text, the third the copy and link times for that text, (these are shown together, as copy times are very small, less than 1% of the total) and the remaining columns show the total times taken with various numbers of open files, from 0 through 100.

service	size (K)	copy + link time	0 open	1 open	10 open	100 open
IDE	10.5	13.0	48.2	47.7	47.7	47.7
FAT	24.9	58.4	118	129	131	135
file	6.2	10.3	17.2	19.1	19.2	20.0

Table 3. Service migration times (ms)

These results show that the various components of the time required to migrate a service can vary substantially between services. While the copy time is small for each (and linear, based on the size of the service, as would be expected), the relative times required to link or initialize the service can be highly variable. The FAT filesystem takes a relatively long time to link, as it makes use of many different library modules, while the IDE driver uses proportionally more time in initialization, as it must create

a thread to process interrupts, and also reads the disk. Both the file services show a small, linear, overhead with the number of open files, while the IDE time does not vary with this factor, as it has no state associated with its clients. The most important conclusion that can be drawn is that migrating services can be done quickly enough that there is little, or no, effect on the time perceived by a user for a service operation..

	operation	above "file"	below "file"	original
"cold"	Open	22395	22299	249
	Read 1K	540	527	9482
	Write 1K	1586	1593	201
	Close	3437	3387	3591
"warm"	Open	462	462	253
	Read 1K	385	383	199
	Write 1K	573	572	184
	Close	7477	7503	9897

Table 4. Compressed File System Operation Times (separate domains)

The final set of experimental results are for the dynamic insertion of the compressed file system service. We repeated the measurements from the first set of experiments, with the compressed file system interposed in two different places: between user applications and the "file" service, and between the "file" service and the FAT filesystem. We took two measurements in each case, the first with all services located in separate domains, and the second with all colocated in the kernel. The results are shown in Tables 4 (separate domains case) and 5 (in-kernel case). For comparison purposes, both tables show the equivalent times (from Table 1) for these operations, without the presence of the compressed file system.

The results from the compressed file system experiments show that opening a file for the first time is very expensive, while the subsequent operation (read) is much cheaper. This is because the compressed file system has to preprocess parts of the file when it is opened, requiring relatively expensive disk reads. However, this is partially compensated for by the subsequent read, which is faster as the information required is already in the buffer cache. The "warm" times show that each of the open, read and write operations are somewhat more expensive, due to the extra processing and CPU time involved in compression, but that this is offset by the reduced disk I/O times (as seen by close). This is a common feature of compressed file systems, where the CPU time for compression must be offset against the frequency of I/O operations that need to

go to disk, and the requirement for less disk space usage.

The important demonstration made by all of the experiments described is that dynamically and transparently reconfiguring the system in order to increase performance, or to include new or changed functionality, is both possible and practical.

	operation	above "file"	below "file"	original
"cold"	Open	23277	23270	167
	Read 1K	130	127	9077
	Write 1K	324	330	94
	Close	3129	3131	3413
"warm"	Open	177	177	122
	Read 1K	106	103	72
	Write 1K	282	280	72
	Close	10654	10687	10958

Table 5. Compressed File System Operation Times (in-kernel)

6 Related Work

Systems have been structured with microkernels for over a decade [1, 17]. One of the primary concerns with such systems is that of performance – the overhead imposed by an IPC/RPC layer will always prevent such systems from performing as well as a monolithic system. This is true, even for such highly developed and optimized microkernels as L4 [11]. To defer this cost, some systems let servers be co-located, gaining performance by avoiding expensive address space switches, and/or eliminating the IPC altogether [2]. Kea however is unique in its ability to dynamically reconfigure and replace services while the system is running, including into the kernel itself. This allows Kea to maintain the benefits of modular structuring and user level development, while also letting the end users of the system have the full performance of a monolithic kernel.

The majority of modern operating systems allow for the dynamic insertion of selected services, usually device drivers or file systems, into the kernel. Typically however, these systems can only be inserted once, and can run only in the kernel context. Unlike Kea, there is no provision for the dynamic replacement of an existing service, for the development of these services in a user-level domain, or for their migration.

Several other systems, notably SPIN [4], Vino [16] and Exokernels [7] have been developed to support application insertion of code into the kernel in order to enhance an application's performance. For trusted code, the Kea mechanisms already described are sufficient to accomplish

this task. In addition, we believe that the vast majority of applications do not need such support, and that those that might benefit from it most, such as database servers [19], are already often trusted by the system anyway. A possible exception to this is an application such as advanced multimedia that needs direct access to the system hardware to gain acceptable performance. In this case, we believe that it is essentially a software design issue, where the services offering access to the hardware need to be designed correctly in the initial system. Of course, Kea allows the user to extend the system, running services in separate domains from the kernel, which eliminates many security concerns, but does introduce a small overhead for the IDC. We believe that this overhead is small enough (typically a few 10's of microseconds) compared to the work actually performed by any real service, that this will be an acceptable solution for many such applications.

7 Conclusions and Future Work

We believe that Kea's design, and the system techniques used in its implementation, provide a practical means of combining the best of both the microkernel and monolithic operating system design philosophies. Kea encourages the modular design of operating system services by requiring various services to have well defined interfaces. It allows these services to be run, tested, and debugged in separate address spaces, facilitating their development. Finally, it provides a means through which these services can be migrated into the kernel, or colocated in a single address space, in order to obtain the best performance. Our measurements show that at least in the file system domain, Kea's performance is comparable to that of a mature operating system.

Kea also allows for some amount of fault-tolerance in the system. Faulty or outdated services can be dynamically replaced, using the same facilities as are used for service migration. Individual services can be run in separate domains, ensuring that any erroneous behaviour on the part of these services is confined to a single domain.

Finally, Kea provides a degree of application specificity. By allowing applications to dynamically replace services on a domain specific basis, the system can be tuned to give the best performance for each application. Although we have not concentrated on this aspect of Kea in this paper, we intend to investigate it further in the future. We also intend to investigate dynamic protocol reconfiguration in the context of the *x*-kernel [12], which we are using to provide networking for Kea.

8 References

[1] M.J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian Jr. and M. W. Young. Mach: A New

Kernel Foundation for Unix Development. In *Proc. Summer USENIX Conference*, July 1986, pp. 93-113

[2] N. Batlivala, B. Gleeson, J. Hamrick, S. Lurndal, D. Price, J. Soddy and V. Abrossimov. Experience with SVR4 over Chorus. In *Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992, pp. 223-241

[3] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX File System. In *Proc. Winter USENIX Conference*, February 1988, pp. 267-275

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers and C. Chambers. Extensibility, Safety and Performance in the Spin Operating System. In *Proc. of the 15th SOSP*, December 1995, pp. 267-284

[5] N. Bhatti and R. D. Schlichting. A System for Constructing Configurable High-Level Protocols. In *Proc. SIGCOMM '95*, August 1995, pp. 138-150

[6] J. S. Crane, *Dynamic Binding for Distributed Systems*. Ph.D. Thesis, Imperial College, May 1997

[7] D. Engler, M. F. Kaashoek and J. O'Toole. Exokernel, An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th SOSP*, December 1995, pp. 251-266

[8] B. Ford and J. Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. *Proc. Winter USENIX Conference*, January 1994, pp. 97-114

[9] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. *Proc. of the 14th SOSP*, December 1993, pp. 80-93

[10] G. Hamilton & P. Kougiouris. The Spring Nucleus: A Microkernel for Objects. *Proc. Summer USENIX Conference*, June 1993, pp. 147-159

[11] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg and J. Wolter. The Performance of μ -Kernel-Based Systems. In *Proc. of the 16th SOSP*, October 1997, pp. 66-77

[12] N. C. Hutchinson and L. L. Peterson. Design of the *x*-kernel. In *Proc. SIGCOMM '88*, August 1988, pp. 65-75

[13] D. Jewett. Integrity S2: A Fault-Tolerant Unix Platform. In *Proc. of the 21st International Symposium on Fault-Tolerant Computing Systems*, June 1991, pp. 512-519

[14] K. Krueger, D. Loftesness, A. Vahdat and T. Anderson. Tools for the Development of Application-Specific Virtual Memory Management. *Proc. OOPSLA '93*, 1993, pp. 48-64

[15] J. Rees, P. H. Levine, N. Mishkin and P. J. Leach. An Extensible I/O System. In *Proc. Summer USENIX Conference*, June 1986, pp. 114-125

[16] M. Seltzer, Y. Endo, C. Small and K.A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *OSDI '96*, October 1996, pp. 213-227

[17] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Léonard, S. Langlois and W. Neuhauser. The Chorus Distributed Operating System. *Computing Systems*, 1(4), Fall 1988, pp. 305-370

[18] P. W. Smith. *The Possibilities and Limitations of Heterogeneous Process Migration*. Ph.D. Thesis, Computer Science Department, University of British Columbia, October 1997

[19] M. Stonebraker. Operating System Support for Database Management. *CACM*, 24(7), July 1981, pp. 412-418.

[20] A. C. Veitch and N. C. Hutchinson. Kea – a Dynamically Extensible and Configurable Operating System Kernel. *ICCDs '96*, May 1996