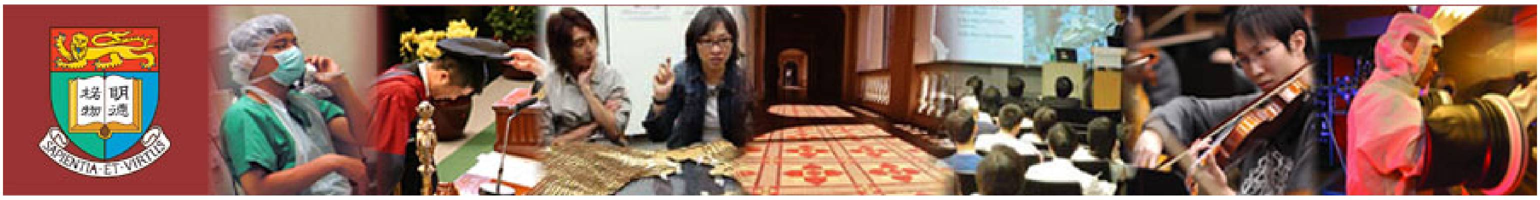


Title	A comprehensive lightweight inter-domain procedure call mechanism for concurrent computations
Author(s)	Loong, AHS; Cheung, WH
Citation	Frontiers of Computer Technology, the 10th IEEE Region 10 Annual International Conference Proceedings, 22-26 August 1994, v. 1, p. 50-55
Issued Date	1994
URL	http://hdl.handle.net/10722/45556
Rights	Creative Commons: Attribution 3.0 Hong Kong License

A Comprehensive Lightweight Inter-Domain Procedure Call Mechanism for Concurrent Computations

Anthony H.S. Loong and W.H. Cheung
Dept. of Computer Science, The University of Hong Kong

Abstract

Many inter-domain procedure calls have been developed to provide fast communication services between protection domains. Different techniques have been employed to trade protection for performance. However, few studies have been made to discuss issues for constructing a comprehensive and generally usable inter-domain procedure call (hereafter as IDPC) facility. In this paper, we evaluate the tradeoff between protection and performance in a IDPC facility, and introduce a new inter-domain procedure call mechanism. Our IDPC mechanism shows its merits on achieving the comprehensiveness with secure protection and good performance comparable to some well-known mechanisms.

1 Introduction

The notion of process plays an important role in contemporary multi-tasking operating systems. Definitions of process and its components vary among different operating systems, but that in Unix is recognized and well-accepted by most people. Unix process, residing within a single *address space* and containing only one *thread of execution*, provides a simple and secure environment to run a single *program image*. However, in some applications, users find that such a strongly protected environment imposes performance burden to inter-process communications, especially in those applications in which protection requirements are loose and process communications are frequent. To cope with this, lot of works have been done in improving the performance of inter-process communications, mostly by loosing the protection barrier between caller and callee domains. Unix's light-weighted thread package, lightweight remote procedure call [1], user-level interprocess communication [2], anonymous RPC [3] are all examples with different extents of trading protections for performances. So, what is the appropriate tradeoff between protection and performance for an inter-domain communication facility?

The answer to this question is not unique. It depends on the generality of the communication service and on the structure of its underlying operating system. In recent years, the concepts of *micro-kernel* and *reconfigurability* have become important treads in operating system design. Famous operating systems like Mach, Chorus, and Amoeba more or less have their kernel components well-encapsulated and modularly constructed. An inter-domain communication facility, being a communicating mechanism which tightly couples with its operating system components, should be designed in such a way that the kernel modularity can be maintained. Moreover, it should be available to application users, not just be used in the system communications only. Thus, design issues like *service semantics*, *naming problem*, and *abnormal exception handling* should be re-considered with *user-level concern*. In fact, by exploring improvements in the aspects of modularity and usability in addition to the basic requirements of *protection* and *performance*, we are adding a new tradeoff consideration in inter-domain communication development: *comprehensiveness*. Issues for achieving comprehensiveness may range from low-level ones such as specifying interactions and managing kernel modules, to high-level ones such as defining calling semantics, resolving names and ensuring robustness. Few studies on analyzing these issues have been made and one major purpose of our work is to fill up the inadequacy.

Among different kinds of inter-domain communication facilities, we focus on discussing the inter-domain procedure call (IDPC). Procedure call itself denotes a simple, clear, well-known and powerful mechanism to transfer data parameters and execution control to a specific destination address. Depending on the location of the address, the execution will result in a local, inter-domain or remote procedure call. This variability, together with its simplicity and effectiveness, makes it a suitable tool in building distributed applications and the basic primitives to construct other distributed facilities. In this paper, we have compared some common inter-domain procedure call mechanisms and found out how they behave in making compromise on protection, performance and comprehensiveness.

To testify the feasibility of our analysis, we have designed and implemented a new IDPC facility which demonstrates a good tradeoff among *good performance*, *secure protection*, and *appropriate comprehensiveness*. The facility is general and reliable enough for those kinds of applications which require fast communications between protected domains, such as network protocol software development where protected boundaries for individual protocols and low cross-protocol overheads for transferring large data packets are necessary[7].

In the following presentation, Section 2 gives the tradeoff survey concerning protection, performance, and comprehensiveness, Section 3 mentions the basic features and the implementation issues in our IDPC facility, Section 4 analyzes the performance, and finally, Section 5 concludes our discussion.

2 Trade-off Survey on IDPC

2.1 Protection and Performance

Trade-off study between protection and performance for inter-domain facilities has been going on for years. Overhead raised in crossing boundary of protection domain affects the performance, while accessibility between the caller domain and the callee domain determines the degree of protection. By varying the degree of protection, performance can be made better or worse. This variability is possible because there are three parameters which can be altered in designing an IDPC mechanism. They are (1) *the different definitions of a protection domain*; (2) *the different ways to ensure its security*; and (3) *the different degrees of protection enforcement.*

Depends on different situations, a protection domain may be defined as a physical host where protection is enforced by hardware boundary; as an address space where protection is enforced by address accessibility; or as a program segment where protection is enforced by hardware segmentation. All these are examples having high degree of protection enforcement in which an execution flow cannot access the area outside its own protection domain. Guaranteed by hardware mechanism to prevent overlapping of data access among protection domains, it results in an *absolute* protected environment in which intentional access to other domains without kernel involvement is impossible. Unfortunately, this involves relatively high cross-domain overheads due to several operations:

(1) *kernel traps*: traps are necessary to transfer control to kernel to perform privileged works, both in the procedure calling and returning operations;

(2) *parameters copying*: transfer of parameters between domains should be accomplished by data copying. Problems will be deteriorated if reference parameters are included since pointers may point to structure which may also contain pointers;

(3) *context switching*: registers should be saved and reloaded, the invalidation of virtual memory cache will further degrade the performance;

(4) *scheduling*: if different threads are responsible for the execution flows in the caller and callee domain, scheduling of threads is necessary during each domain crossing.

All these overheads must appear in a remote procedure call but optimizations can be applied to avoid some of these, if the procedure calls just happen between domains resided in a single host. In fact, some IDPC mechanisms explore optimization further by tolerating the possibility of allowing threads to access locations outside their protection domains. Usually, this occurs in the process of executing a procedure where partial area of the caller domain is shared with the callee domain. A common technique is to use shared memory to hold parameters and return values, hence reducing the overhead of data copying. However, this opens the possibility of having illegal accesses from the peer domain. In the case where threads are directly transferred from the caller to callee domain, the argument stack or the thread's whole stack can be shared to eliminate data-copying but information in the stack is then vulnerable to unexpected or intentional invalid access. Thus, although an inter-domain call with loose protection enforcement will have a better performance than those with absolute protection enforcement, they should trust their applications for not making attempts to leak through these extra shared memory between the domain boundaries. To alleviate the situation, some implementation will introduce specific language constructs to restrict programming.

2.2 Comprehensiveness

Compromising protection is not the only way to gain performance in constructing an IDPC mechanism. Very often, special techniques will be used in an implementation to reduce overheads, but they may have the adverse effect of imposing some forms of restrictions on its applications. For example, in transferring large amount of data, the technique of remapping the whole memory page holding the data can reduce a considerate amount of data copying overhead because it requires only changing the page table directory entries. However, as the granularity of data transfer is in page size, careful page alignments on the transferring data should be made to prevent from sharing of unrelated data. Consequently, cross-domain communication services adopting this technique should employ special data structures and operations to ensure that page alignment is properly made. Moreover, with specific data structures, they suffer from less efficient and restricted use of memory address spaces. This is an typical example of trading generality for performance

Comprehensiveness concerns the improvements on generality and usability of an inter-domain facility, based on the basic assumption of maintaining kernel modularity. To ensure comprehensiveness in an IDPC mechanism, more issues other than that supporting normal operations or enchancing performance and protection should be considered. Generally speaking, each issue will relate to a supporting facility or a specific aspect which may not be essentially necessary in the basic calling operations of the IDPC mechanism , but must be provided when the facility is generally used. Some of these issues are summarized as follows:

Name Binding:

A naming mechanism may be avoided if the caller domain has a static knowledge of the names of all other callee domains. However, for an inter-domain facility that is available to general users, such knowledge is absent. So, a naming mechanism should exist to allow callee domains to export their procedures and caller domains to import or bind these procedures. Amount of overheads added in the calling operation depends on where and how the binding information is kept and retrieved. If this information can be kept in the user address space, the loss of performance due to name binding can be reduced.

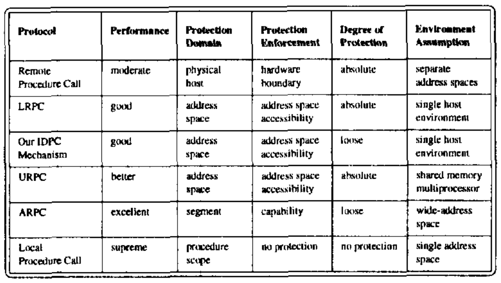
Thread and Address Space Management:

Within a modular operating system, threads and address spaces are modularized as individual objects from which process model and thread packages are constructed. If an IDPC mechanism is allowed to apply on both user-level domains and kernel-level domains, it is inevitable to have its implementation carefully coupled with these operating system components. In order to maintain modularity, the management of these objects and the integration of the mechanism with the overall kernel structure should be careful made. Nevertheless, running a comprehensive IDPC mechanism under a modular operating system environment should incur extra performance overhead because this requires a more strictly, well-encapsulated, and sophisticated method to keep and update the state information of the system objects.

Robustness:

If the mechanism is constructed for specific application, abnormal behaviors can be easily anticipated and corresponding control or assumption can be made to avoid their occurrences. Unfortunately, it is not the case for general use. Due to misuse or ignorance by users, fatal exceptions such as jumping to invalid or unexisted domain, calling procedure without importing, possible deadlock during exporting and importing procedures, and sudden termination of threads during calling, may result. They should be handled to ensure the robustness of the mechanism. However, significant overheads may be imposed since extra checking statements should be added to the implementation and they must be performed in every cross domain operation. In spite of this fact, robustness is an inevitable issue in comprehensiveness since it determines the usability and the reliability of the mechanism.

2.3 A Comparison on Some IDPC



Protocol	Performance	Protection Domain	Protection Enforcement	Degree of Protection	Environment Assumption
Remote Procedure Call	moderate	physical host	hardware boundary	absolute	separate address spaces
LRPC	good	address space	address space accessibility	absolute	single host environment
Our IDPC Mechanism	good	address space	address space accessibility	loose	single host environment
URPC	better	address space	address space accessibility	absolute	shared memory multiprocessor
ARPC	excellent	segment	capability	loose	wide-address space
Local Procedure Call	supreme	procedure scope	no protection	no protection	single address space

Table 1: Features of Different IDPC Mechanisms

Table 1 shows a comparison of the features of some well-known IDPC mechanisms. On the top of the table stands the remote procedure call [5] which achieves the best possible protection enforcement since it has no way to access memory directly on a remote machine. Derived from a remote procedure call mechanism, LRPC (lightweight remote procedure call) [1] is the initial attempt to improve performance of on RPC happened between local domains. It represents a category of those inter-domain procedure calls which make uses of a single thread to finish the whole procedure calling operation. Stack segment is attached to the thread and protection is ensured by page mapping and hardware segmentation. Compared to conventional lightweight threads, this kind of threads are middle-weighted because overhead of switching between address spaces is still required, thus hindering performance improvement to some limit. Therefore, some researchers think of confining the execution of a thread in a single address space. URPC (user-level remote procedure call) [2] takes advantage of shared-memory multiprocessors so that thread contexts can be redundantly placed in different processors. Each processor is responsible for only one address space so context switching of threads does not require switching of address spaces. However, this implementation trick of taking advantage from multi-processor environment makes it unfair to compare with other IDPC mechanism described.

As 64-bit machines are available, using a single address space to hold all threads become possible and reasonable. ARPC (anonymous remote procedure call) [3] assumes sparse use of such a 64-bit wide-address space. It lowers the degree of protection enforcement by making use of anonymous addresses as capabilities. Accesses can be made possible only if the caller can get the right capability. Though the chance of guessing the capability may be low, it is still possible. In other word, such kind of inter-domain procedure call mechanism indeed gains very good performance improvement by sacrificing absolute protection enforcement. Moreover, ARPC has different versions which requires different degrees of trust on the application users. The greater the degree of trust is, the better the gain in performance but the larger the restriction on applications is. This shows an example of compromising comprehensiveness and protection for performance. At the extreme case, local procedure call do not enforce any protection, resulting in the best performance.

Besides inter-domain procedure call, other cross-domain communication mechanism like Fbufs [4] and Shipping Container [8] have been designed. As mechanisms applied to particular areas, they make assumptions on the operating environment that limits their generality. In the following section, it can be seen how our IDPC mechanism is designed and implemented comprehensively under a modular operating system environment, with performance close to these application-oriented facilities.

3 A New IDPC Facility

In order to explore the issues of comprehensiveness that has been weakly emphasized in the previous works, we have constructed a new IDPC facility which demonstrates how these issues can be tackled in simple yet efficient way without heavy kernel mediation. Despite that the facility is available to user, its novel feature of allowing kernel threads to cross address space boundaries offers an opportunity for users to flexibly build applications with concurrent executions in an environment where separate protection domains are distributed. Here, we describe some of the special features in our facility:

Simplicity and generality: it provides a familiar, simple but yet generic user-level communication utility to support concurrent applications with multiple domains.

Simple declaration: naming of domain, declaration of export and import procedures are specified in a special definition file. No extra amendments on the program is needed when defining and calling an inter-domain procedure, thus it makes the mechanism transparent to the original program codes.

Dynamic binding: in addition to static binding of procedures between caller and callee domain, run-time binding is also possible. It allows the callee domain to be changed dynamically. Such characteristic enables applications with run-time reconfigurable servers can be built to ensure service flexibility and reliability.

Intra-domain call: binding of exported procedure to the same domain is possible, allowing it to support both inter-domain and intra-domain communications transparently.

No kernel-mediated binding: all binding information is kept and retrieved in the caller or callee own address space without kernel involvement. This favors better performance and less interference to kernel structure and modularity.

User-level name server: name server is implemented as a separate module running in a user address space. For the sake of consistency, its communications with other domains are also done through the same

IDPC mechanism. Moreover, to eliminate unnecessary overhead, the name server will never be involved during a cross-domain call operation.

Single thread involvement and mappable stack: to reduce context switching overhead, the same thread that calls an inter-domain procedure in its caller domain will be assigned to execute the procedure definition in the callee domain. Closely adhered to the thread, the whole stack will be mapped to the callee domain so that copying of parameter data can be eliminated. In addition, dynamic stack and shared memory allocation utility is built to facilitate pointer parameter passing.

Modular development: our operating system is dedicately constructed to provide a modular abstraction which includes primitives manipulating on the system objects like thread and address space. Based on this architecture, the IDPC mechanism can be developed without ad-hoc kernel modifications. This greatly increases the portability of the mechanism to other modular operating systems.

The facility is built on a locally-modified version of Minix operating system [6] which incorporates a virtual memory system and a preliminary modular kernel that comprises thread, address space and program objects. The following sections will give a more detailed description about our IDPC implementation.

3.1 Thread and Address Space Management

In our development, each thread represents a typical execution flow while each address space denotes a protection domain holding a single code image loaded from a program object. The relationships between thread, address space and program object are illustrated in Figure 1. When a program object is created, memory segments are allocated to load the text and data code of the program image. It can then be attached to an address space object by mapping the text and data segment to the top of the address range. Stack segment, on the other hand, is allocated only when a thread object is created. When thread object is latter attached to an address space object, specific address range will be allocated to where stack segment is mapped. Moreover, this address range also includes a new data segment the content of which is copied from the loaded data segment of the program object. This data segment in fact represents the initial value of the program data. To enchance concurrency, more than one thread can be attached to an address space. As a result, by varying segment register during context switching, all threads in an address space can use the same text segment but will have their own copies of data and stack segments. During an IDPC operation, thread will be jumped to the callee address space and stack segment will be mapped to the destinated address space, hence saving time for copying parameters. However, the memory segment for the stack is actually functionally divided into two areas. The top one stores the normal stack data while the bottom one holds dynamically allocated data. This data region is private to individual threads so that a thread can access it no matter in which address space the thread is in. This provides a mean to handle reference parameters efficiently if data is kept in these data regions. In addition, to facilitate sharing of large data structures, all address spaces contain a shared memory segment where all threads from different address spaces can access.

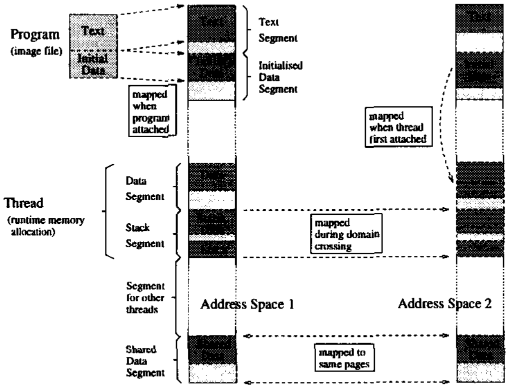


Figure 1: Program, Thread Memory and Address Space

As a utility for users, our system offers a simple set of user-level routines to create and manage these thread and address space objects. Memory utility is also provided to manipulate shared memory and dynamically allocated stack data. However, all these routines are built on top of the kernel-level primitives to avoid serious distortions to the existing kernel structure and components.

3.2 Name Binding

Alike the constructions of any RPC mechanisms, one major issue of building an IDPC facility is to resolve or map names between the caller and the callee domain. Such a name binding usually involves two phrases: *exporting* and *importing*. The exporting phase requires callee to export those procedures that can be called from other domains. The importing phrase allows the caller to bind procedures from the appropriate callee, so that the caller can use the procedure definitions in the callee domain.

In our implementation, all bindings are done through a name server which provides general services of maintaining mappings between identifiers. To support our IDPC mechanism, the name server needs to keep three types of mappings:

(1) The first mapping is an one-to-one mapping from symbolic domain name to a unique address space identifier.

(2) The second mapping is another one-to-one mapping from address-space identifier and symbolic procedure name to a unique procedure identifier. This keeps a unique identification for all exported procedures in an address space domain.

(3) The last one is a mapping from address-space identifier and its procedure identifier to the corresponding symbolic procedure name. It helps in cleaning up entries in other mappings when an address space domain is removed.

For a program, no matter it is a caller or a callee, or both, a user should prepare an export/import definition file which specifies the symbolic name of its address space domain, its exported procedures, and its imported procedures with the names of domain from which they are imported. Prior to running a program image, exported procedures will be uniquely identified, and related name information will be automatically sent to the name server. After this, procedures can be imported by retrieving name information from the name server and storing them in a reserved area of the user address space. This area composes of two tables: ***export procedure table*** and ***import procedure table***. Export procedure table keeps the actual address of each exported procedure. Searching this table with a procedure identifier as an index, a thread can immediately be jumped to the exact location of the exported procedure. The import procedure table keeps the procedure identifiers of the imported procedures and the address space identifiers of the callee domain. This allows a fast retrieval of binding information in the midst of domain crossing without involving the name server. Deadlocks between export and import operations will not occur since all exports of procedures are done before imports and the attempts to import will be ceased to continue after a fixed limit of trials. Figure 2 shows an illustration of the exporting and importing operations discussed.

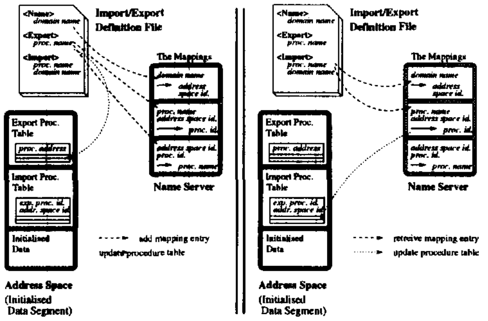


Figure 2: Exporting and Importing

Our facility also includes a system call which allows binding of procedures at run-time. This increases programming flexibility since the definition of a procedure can be changed at run-time by binding it to a new callee. Moreover, the name server communicates with other domains using the same IDPC mechanism through a set of pre-exported procedures. This eases modification and replacement of the name server.

3.3 Calling

Invoking an IDPC call involves transferring a thread from an address space to another. During a procedure call, the stack will follow the thread to move from the caller to the callee address space, and return after procedure execution. The technique used is to set up the stack frame in such a way that before the actual execution of the procedure in the callee domain, the stack frame is identical to that of a typical local procedure called in the callee domain. Refer to Figure 3, the calling mechanism will undergo the following operations:

(1) Calling an imported procedure is actually calling a special system call which requires a stack instance as that shown in *Stage 1*. The top entry keeps the returning address; the second keeps the exported procedure identifier; excluding the two dummy entries followed, the remaining entries are used to hold the arguments.

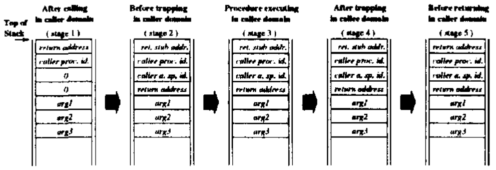


Figure 3: Procedure Calling

Such setup can be made because before actual compilation, all statements of calling imported procedure will be scanned and automatically replaced by this system call with appropriate parameters added. This job is done by a ***pre-compilation transformation program*** which reads both the program source code and the export/import definition file.

(2) In the system call, stack frame will be updated to include the caller address space identifier. The original return address will be lowered and replaced by a fixed address of a special library routine called *returning stub* (*Stage 2*).

(3) Read from the import definition table, the callee's address space identifier can be found and it will become one of the parameters in the kernel trap.

(4) Trap to the kernel, program counter will be changed to point to another library routine called *calling stub*. The thread is now ready to jump to the callee domain. Pages are appropriately mapped, related virtual memory registers are set and finally the trap is returned.

(5) The thread is now in the new domain with program counter pointed to the *calling stub*. By using the procedure identifier stored in the stack as an index to the export procedure table, the stub can simply look up the address of the exported procedure and jump to it. Now the stack instance (*Stage 3*) is similar to that of a local procedure call except the returning address is set to the address of the *returning stub*.

(6) On returning from the callee's procedure, the returning stub will be called. The returning stub will invoke another kernel trap (*Stage 4*).

(7) By looking at the caller address space identifier stored in the stack frame, thread can be switched back to the caller address space. The top of the stack is changed back to the original returning address and the program counter is set to another library routine called *cleaning stub* (*Stage 5*) which just executes a *return* instruction. Since the size of the stack have not been changed during the whole operation, the stack instance can be cleared up in the return operation like a local procedure call does.

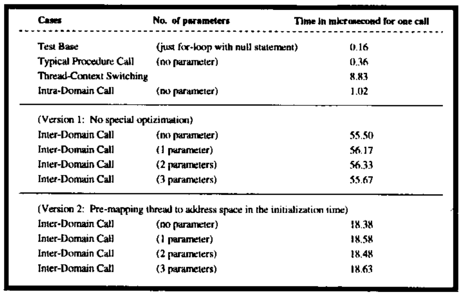
3.4 Robustness

Checking statements and data structures have been added to our implementation to keep track of error conditions. A call will return without executing if invalid binding information is found or the address space of the callee does not exist. Premature termination of a thread is properly handled and removal of an address space with active threads is not allowed.

4 Performance and Analysis

Performance is done by taking the average time of making 100000 call operations running on a IBM 486-DX PC. Besides testings on basic overheads like typical procedure call, thread context switch and intra-domain call, two versions of our implementations are tested. Version 1 is a non-optimized implementation whereas Version 2 uses a pre-mapping technique which is done during the creation of thread. It eliminates some page mapping overheads during an IDPC operation. Table 4 shows the result. It should be noticed that all data shown include the test loop overhead.

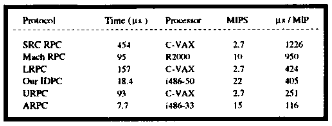
Several points should be noticed. First, our intra-domain call has achieved a good performance. It only involves some operations of name binding and error checks. Since the binding information is kept in the caller address space, no kernel involvement is necessary. Second, increasing the number of parameters has insignificant impact on the performance because we map the whole stack in which parameters are placed. Third, the large difference in performance between version 1 and version 2 is due to the overheads in writing up the page table entries. However, the pre-mapping technique used in version 2 requires software modules and resources in kernel to be arranged in a specific manner, so we have actually traded some flexibility for performance. Finally, it can be seen that the total time for an inter-domain call (18.38 ms) is nearly equal to the time for an intra-domain call (1.02 ms) plus two context switchings (8.83 ms × 2, one for calling, and one for returning). This shows that our mechanism involves very little overhead because most of the overheads come up from the context switching operation, which is determined by the kernel, not by the mechanism itself. This switching overhead includes time taken for making a kernel trap, saving and loading thread context, and updating virtual memory registers. If kernel optimizations are made on this, the overall performance of our mechanism can be further improved.



Cases	No. of parameters	Time in microsecond for one call
Test Base	(just for-loop with null statement)	0.16
Typical Procedure Call	(no parameter)	0.36
Thread-Context Switching		8.83
Intra-Domain Call	(no parameter)	1.02
(Version 1: No special optizimation)		
Inter-Domain Call	(no parameter)	55.50
Inter-Domain Call	(1 parameter)	56.17
Inter-Domain Call	(2 parameters)	56.33
Inter-Domain Call	(3 parameters)	55.67
(Version 2: Pre-mapping thread to address space in the initialization time)		
Inter-Domain Call	(no parameter)	18.38
Inter-Domain Call	(1 parameter)	18.58
Inter-Domain Call	(2 parameters)	18.48
Inter-Domain Call	(3 parameters)	18.63

Figure 4: Performance Data

In comparing our implementation with other IDPC mechanisms, we have achieved a performance which is quite close to that of LRPC. Figure 5 shows the performance data of executing a null call in some IDPC mechanisms (data are taken from Table 2 in [3]). However, our implementation do not have an absolute protection enforcement since the whole stack is visible in the callee domain. The situation can be alleviated since we rely on compiler to avoid generating codes in typical procedure that will access data in stack other than its local variables and parameters.



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
Our IDPC	18.4	i486-50	22	405
URPC	93	C-VAX	2.7	251
ARPC	7.7	i486-33	15	116

Figure 5: Performance of a Null Call in Some IDPC

5 Conclusion

This paper presents a study on various inter-domain procedure call mechanisms. A survey has been conducted to evaluate different ways of compromising protection for performance. In addition, we discuss the issue of comprehensiveness which is essential in constructing a general and useful IDPC mechanism.

A new IDPC facility has been built to validate our study. Apart from its good performance and protection, it incorporates necessary facilities to support comprehensive IDPC services - name binding, thread and domain management, and robustness. The technique of mapping the whole stack segment and using dynamically allocated data improves performance in transferring large data. Future works include using this facility to develop some sophisticated concurrent programs such as construction of network protocol software. The facility can also be extended to accommodate remote procedure calls and to build system components for a reconfigurable operating system.

References

[1] B.N. Bershad, T.E. Anderson, E.D. Lazowaska, H.M. Levy, "Lightweight Remote Procedure Call," ACM Trans. on Comp. Systems, Vol. 8, No. 1, Feb 1990, pp 37-55.

[2] B.N. Bershad, T.E. Anderson, E.D. Lazowska, H.M. Levy, "User-Level Interprocess Communication for Shared Memory Multiprocessors," ACM Trans. on Comp. Systems, Vol. 9, No. 2, May 1991, pp 175-198.

[3] C. Yarvin, R. Bukowski, T. Anderson, "Anonymous RPC: Low Latency Protection in a 64-Bit Address Space," Proc. of 1993 Summer USENIX, Jun 1993, pp 175-186.

[4] P. Druschel, L.L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," TR, Dept. of Comp. Sc., U. of Arizona.

[5] A.D. Birrell, B.J. Nelson, "Implementing Remote Procedure Call," ACM Trans. on Comp. Systems., Vol. 2, No. 1, Feb 1984, pp 39-59.

[6] A.S. Tanenbaum, F. Meulenbroeks, R. Michiels, J. Muller, J. Pickert, S. Reiz, J.W. stevenson, "Minix 1.5 Reference Manual," Prentice Hall.

[7] A.H.S. Loong, W.H. Cheung, "An Implementation Model for Developing Network Protocol Infrastructure," Proc. of the 1993 IEEE Region 10 International Conf. on "Computers, Communication and Automation," Oct 1993, Vol. 1, pp 519-522.

[8] J. Pasquale, E. Anderson, P.K. Muller, "Container Shipping: Operating System Support for I/O-Intensive Applications," IEEE Computer, Vol. 27, No. 3, Mar 1994, pp 84-93.