Shared Memory Support for Object-based RPC

René W. Schmidt[*], Jeffrey S. Chase[†] and Henry M. Levy

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

This paper describes object-based runtime support for efficient access to protected objects, i.e., objects belonging to server programs that export protected services to untrusted clients. Modern operating systems use hardware-based protection domains to protect a server's objects; clients access those objects through cross-domain (RPC) calls to methods that execute within the server's domain. Unfortunately, exclusive use of RPC sacrifices performance in exchange for protection.

We show how to provide safe and efficient access to server data by client read-sharing of some server objects and methods. Local client calls to read-only methods execute directly in shared memory, without the cost of protection crossing. We introduce a version-based mechanism to synchronize the readers with write-methods executing in the server domain. Object-based proxies hide the use of shared memory from client programs. Measurements of a simple application demonstrate that our techniques can deliver performance close to an unprotected implementation, but with the modularity and server protection of traditional RPC.

1 Introduction

The client/server model is commonly used to provide data sharing and protection in software systems. Client applications access the data through a fixed procedural interface defined by the server. Calls to the server utilize a protected procedure call mechanism, typically *Remote Procedure Calls* [1], to execute code within the server's domain.

In an object-based environment, the client/server model is implemented using object invocation mechanisms, which are the equivalent of RPC. In this case, each call specifies a particular instance of an abstraction supported by the server. Proxies [2] are typically used to implement the server invocations in this case. The protected server objects (i.e., objects that exist inside the server's domain) are represented by a local proxy object in the client. The proxy contains methods (RPC stubs) that forward invocations to the server. This technique makes local and protected objects look nearly identical to the client programmer, and therefore simplifies program development.

The disadvantage of this approach is its high performance cost, due to crossing address space boundaries, validating arguments, and so on. For this reason, many programs execute in a single protection domain rather than in several domains, at a loss of protection and security.

Bershad *et. al.* have shown that the performance of same-machine cross-domain communication can be considerably improved by using shared memory [3]; Lightweight Remote Procedure Call (LRPC) uses a shared stack to avoid copying parameters and return values between protection domains, and reduces scheduling overhead by running the client thread in the server's domain. However, even with LRPC, cross-address space communication is substantially more expensive than a local procedure call.

This paper shows how shared memory can be further exploited to provide safe, protected, and efficient access by clients to server data. The objective is to give clients the performance of a local invocation for most accesses, but with the protection guarantees of a separate server domain. To do this, we introduce the idea of read-only methods, read-only shared proxies, and special synchronization techniques to coordinate cooperating processes. We assume that server access is "read mostly," that is, the majority of calls read but do not modify server data, while a small percentage

[*]Current address is Department of Computer Science, University of Aarhus, DK-8000 Aarhus C, Denmark.

[†]Current address is Department of Computer Science, Duke University, Durham NC 27708-0129

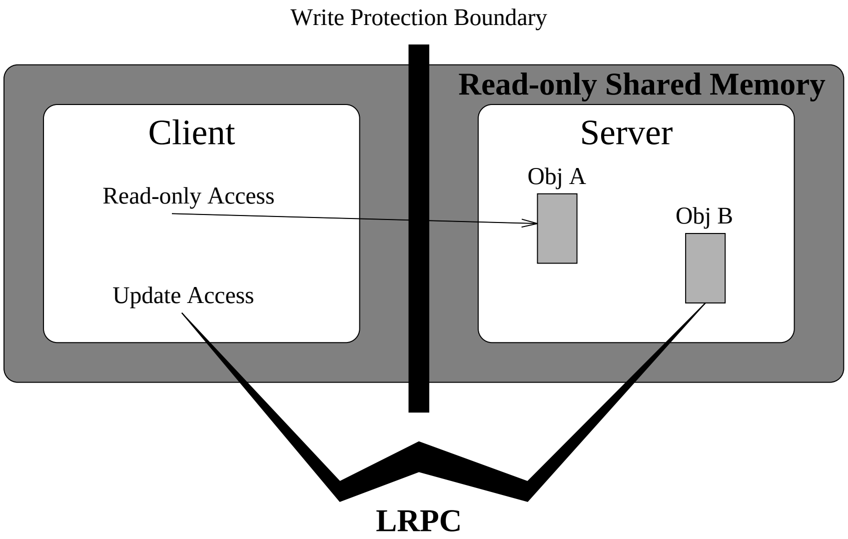

Figure 1: Optimizing Client/Server communication by using read-only shared memory.

of calls change that data. For example, CAD applications have been shown to exhibit such behaviour [4]. A Dexter-based hyper-text system [5] also needs mostly read-only access to the applications that contain the data for each node in the hyper-text document.

Using our techniques, a server maps its data segment read-only into a client's address spaces, thereby allowing the client domain to access (read) the server's data directly. Data integrity is maintained, because protected procedure calls must still be used to modify protected objects. Furthermore, we uses shared proxies to eliminate the overhead of instantiating local proxy objects in each client, and increase parameter passing efficiency. The read-only memory sharing and read-only methods are shown in Figure 1.

The major difficulty with this technique is synchronizing access to the server's data. In particular, one client could be reading and scanning server data, while another client is modifying that data through a method executing inside the server. Because clients do not have write access to the server, they cannot directly acquire or release server locks without performing cross-domain calls, one on acquire and one on release. Such calls would obviously negate the value of using read-only methods. To solve this problem, we have developed a hybrid locking scheme, called version-based synchronization, that permits read-only methods to synchronize *without* executing any store instructions.

We have constructed a prototype implementation of these techniques for the Opal system [6, 7], a single virtual address-space operating system. Our implementation shows how read-only sharing of memory between client and server can be incorporated into an object-based RPC toolkit, so that its use is transparent to the user. While the Opal memory model motivated this work, none of the techniques presented depend on the single address-space model; they can be used on any operating system where applications can coordinate sharing of memory in some manner.

The rest of the paper is organized as follows. The next section summarizes related work. Section 3 presents version-based synchronization. Section 4 outlines the implementation of a generic object-based RPC toolkit. Section 5 and 6 describes how read-only methods and shared proxies can be implemented, respectively. A performance evaluation of the extended toolkit is shown in Section 7. Finally, Section 8 presents our conclusions.

2 Related work

Our work uses memory sharing and synchronization techniques to accelerate the performance of object-based applications executing in separate protection domains. It is thus related to a number of previous research efforts that support object-based programming and those that support fine-grained synchronization. In the following we describe some of those projects and how our work relates to theirs.

The Emerald system [8] was a distributed object system supporting fine-grain object mobility and efficient local performance. To achieve high performance, all objects on a local machine were located in the same protection domain, so access to local objects could be done directly with load and store operations. Data integrity was ensured by the use of a safe language. In our work, we do not require a safe language; instead, we use read-only methods to provide access to server data located on the same machine, while requiring cross-domain calls for methods that modify server data.

Systems such as Eden [9], Clouds [10], Choices [11] and Spring [12] integrated object-based RPC support in the operating system kernel. High performance of cross address-space calls is crucial for such systems. Object-based RPC libraries have been built for a wide variety of programming languages as well, including Modula-3 [13], BETA [14], and C++. Our optimizations are at the runtime level, but we belive that they are general and apply equally well to operating systems.

The proxy implementation in Choices [11] is similar to our shared proxies, but the rationale behind the design is different. In Choices proxies are allocated in read-only memory by the kernel to assure authentication of proxies and to be able to change the implementation of proxies without having to re-compile client programs. Proxies are not shared between clients.

Several experimental operating systems, such as Opal [7] and Monads [15], provide a single shared address space for all applications. Protection and addressing are decoupled in these systems which makes it possible for applications to easily coordinate sharing of data at runtime. Our techniques can easily be utilized on top of such systems (and, in fact, were prototyped on Opal), however they are independent of such single address space support.

Herlihy [16] has provided a basic methodology for implementing non-blocking synchronization and mutual exclusion; he uses two counters (`check[0]` and `check[1]`) to ensure that an object is not copied while it is in an inconsistent state. The two version numbers in version-based synchronization serve a similar purpose, but only read-only methods are implemented lock-free in version-based synchronization.

Should we write something about Time Warp? Software Fault Isolation?

Our work builds in many ways on all of these efforts. We show how shared memory can be used to support high performance (read-only) method invocation between non-trusting applications, without requiring special language support. In addition, we show how method invocations can be synchronized with concurrent, protected, data updates.

3 Version-based synchronization

At the basis of our technique for efficient invocation is version-based synchronization, which we use to control concurrent object access. It has the special property the a critical section that does not update an object can be implemented without executing any store instructions. However, version-based synchronization is a general technique and its application is not limited to object-based RPC. For example, it could beneficially be used to synchronize parallel programs, or to provide applications with direct access to operating system state without a task-switch. In this section we present the details of version-based synchronization; then we describe in the following section the environment in which this synchronization is used.

In contrast to most other synchronization techniques, version-based synchronization is a hybrid between the interlocking and the lock-free technique. A critical section that modifies the state of an object is implemented using an interlocking scheme. A critical section that only reads data from a shared object is implemented lock-free.

Lock-free synchronization is an optimistic scheme; there is very little overhead if concurrent access to an object is rare. All updates execute in parallel and an update is always performed on a copy of the object's data. The update is committed in an atomic operation[1] by updating the object's data pointer to reference the new updated data. If the data pointer was changed since the copy was made, the commit fails and the update has to execute again. Lock-free synchronization is considerably more complex to use than interlocking. However, several authors have shown that lock-free [16, 17] synchronization is both practical and efficient.

Version-based synchronization exploits the asymmetry in the multiple-readers/single-writer synchronization protocol. The inherent concurrency in lock-free synchronization is ideal to permit many readers (r-methods) to execute simultaneously, and the mutual exclusion property among writers (w-methods) is easily expressed using locks.

3.1 Implementation

Figures 2 and 3 show how a critical section is implemented for a w-method and a r-method respectively. The code requires simple assignments (line R3 and R5) and simple increments (line W3 and W5) to be

[1] Typically `load-linked/store-conditionally` or a `swap` instruction are used.

atomic operations. There are no bounds on the execution time for the critical sections (line R4 and W4). The *Mutex* and *Version* variables are shared among all methods for a particular object.

W1: $\triangleright$ *non-critical section* $\triangleleft$
W2: **Mutex.lock()**;
W3: Version[0] $\leftarrow$ Version[0] + 1;
W4: $\triangleright$ *critical section* $\triangleleft$
W5: Version[1] $\leftarrow$ Version[1] + 1;
W6: **Mutex.unlock()**;
W7: $\triangleright$ *non-critical section* $\triangleleft$

Figure 2: Implementation of a read/write critical section.

R1: $\triangleright$ *non-critical section* $\triangleleft$
R2: **do**
R3: $\quad v_1 \leftarrow$ Version[1];
R4: $\quad \triangleright$ *critical section* $\triangleleft$
R5: $\quad v_0 \leftarrow$ Version[0];
R6: **while**($v_0 \neq v_1$);
R7: $\triangleright$ *non-critical section* $\triangleleft$

Figure 3: Implementation of a read-only critical section. v_0 and v_1 are local variables to the method.

Write-methods synchronize among themselfs using a mutex lock. Furthermore, they update the version-numbers. The version-numbers are used by the r-methods to detect concurrent access. Version[0] counts the number of times an update has begun on an object, and Version[1] counts the number of times an update has completed. If the two numbers are equal the object will be in a consistent state, otherwise a w-method is executing.

A r-method can ensure that only consistent data is read by reading the version-numbers in opposite order of which the w-methods update them. First, an r-method reads the number of completed updates into a local variable v_1 (line R3). It then queries the object. When the query is completed the number of times an updates has begun is read into the local variable v_0 (line R5). If v_0 is equal to v_1 a new w-method has not been started since the read-only method started, and the object was thus consistent during access. Otherwise, one or more w-methods have started executing while the r-method queried the object, and the query is re-executed (line R6).

By combining lock-free and interlocking, we get a simpler implementation than any of the two will provide alone; for interlocking, care has to be taken to allow more than one reader to access the object at a time; for the lock-free strategy care has to be taken to implement atomic update for the writers, i.e., we eliminate data copying for write-methods.

The technique extends directly to implementing read-only methods that depend on several locks. Instead of just accessing one set of version-numbers, several sets can be checked and the while-condition (line R6) evaluates to false if one or more of the sets are inconsistent.

3.2 Requirements of read-only methods

With version-based synchronization, an r-method might be reading object data while a w-method is updating that data; the r-method will detect that update and redo its operation. However, because of this concurrent access, certain constraints exist on the code in a critical section for an r-method:

- It has no side effects. The number of times a critical section is executed is unpredictable and the data it is accessing might not be consistent.
- It must be total, i.e., an r-method's behaviour is well-defined on all input. Inconsistent data must not cause a program error.

The efficiency of version-based synchronization depends on how often r-methods are re-started, which in turn depends on the execution time of r-methods and the update rate of objects. The technique is therefore best for objects where reads are more common than writes, and where r-methods execute quickly.

4 Object-based RPC

This section provides the necessary background information on object-based RPC, in order to discuss how an object-based RPC toolkit can be extended to take advantage of read-only shared memory.

For object-based RPC, a call pertains to a particular fine-grained object inside the server. The toolkit allow pointers to these protected objects to be passed as arguments and returned as result when a protected object is called. An object-based toolkit provides the following:

- transparent access to protected objects for clients,
- protected updates of objects for the server, and

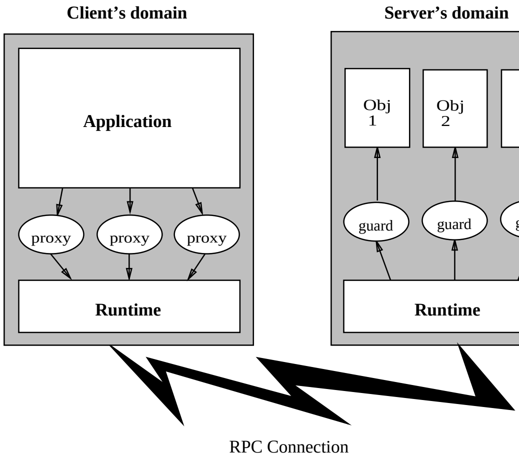

Figure 4: Proxies, guards, and protected objects.

- synchronization of concurrent access to protected objects.

Note that protection of server objects does not necessarily imply complete data hiding. This is a result of our basic requirement for optimizing the RPC protocol, i.e., client methods can read some server data directly without executing an RPC call.

4.1 Implementation

In our system, *proxies* are used to hide the protection boundary from the applications. A high-level view of the model is shown in Figure 4. Protected objects are represented as proxy objects in the clients. A proxy marshalls the parameters and executes an RPC call. On the server side, the RPC call is handled by a *guard* object, which unmarshalls and validates the parameters and executes a local object invocation.

Proxies and guards correspond to client and server stubs, respectively, in a standard RPC system [1]. A pointer to a protected object is passed as a capability [18] between the clients and servers. A capability is an unforgeable name for a protected object. We use password capabilities [19], which are probabilistically rather than absolutely impossible to forge. The capability for an protected object is both stored in the proxy and in the guard.

On the client side, a capability is translated into a proxy pointer by looking it up in a hashtable that maps capabilities to proxies. If the capability is found, the proxy pointer is returned; otherwise a new proxy is instantiated for the server object. On the server side, the guard maps a capability directly to a protected object by examining the information stored in it. The asymmetry is because there is a one-to-one mapping between capabilities and server objects, but a one-to-many mapping between capabilities and proxies.

5 Read-only methods

Read-only methods can be transparently integrated into an object-oriented toolkit. Transparency is maintained, because protected objects are still accessed through a procedural interface. The procedures for read-only methods just invoke the server's code directly. This eliminates the cost of marshalling, cross-domain control transfers, and parameter validation on the server side.

Providing read-only methods does not compromise the integrity of the server application. Protection of server objects is maintained because modifications to server data are done through RPC calls. The server has full control over which modifications can be made to its objects, although the server has lost control over which objects the client can read. For servers that contain confidential information, this technique might not be appropriate. The trade-off is increased performance for the price of weaker access control. However, if the server wishes to hide some of its data, it can use encryption or, more likely, place the secure data in a separate segment that is not made readable to clients.

Synchronization of access to protected objects is done with version-based synchronization, which does not require any store operations to synchronize concurrent r-methods. Read-only methods are supplied by the server and the clients are requried to trust them. A r-method must be total (see Section 3.2) so the method might, in the case of inconsistent data, return an error code or generate an exception. In this case, these would be defined in the server's interface, and the client must be prepared to handle such situations.

Read-only sharing of memory is a further generalization of proxies that cache immutable data to optimize client/server communication. With read-only methods, immutable server data can be read directly from the server, reducing proxy size and copying costs. In fact, no distinction between immutable and mutable data has to be made, because all server data is directly accessed.

5.1 Marshalling

Marshalling of arguments for read-only methods are not necessary, because a method invocation is performed directly. Furthermore, validation of arguments on the server side can be relaxed. Read-only methods are executed directly by the client, so if a client supplies invalid arguments, or follows an invalid pointer, it can only hurt itself; the call is invisible to the server.

However, all pointers that are manipulated by a client are proxy pointers, so the stub code for r-methods must convert proxy pointers passed as arguments to pointers that point directly to protected objects, and vice-versa for pointers that are returned as a result.

Translating proxy pointers is straightforward, because the protected pointer can be stored in the proxy. Unfortunately, a hashtable lookup is still needed to convert a protected object pointer into a proxy pointer. Proxies are local to clients, so the server cannot directly return a proxy pointer, but must still return a capability. In Section 6 a solution to this problem is presented.

```
ObjProxy *ObjProxy::myMethod(ObjProxy *p)
{
   Obj *real_p = p->GetObjPtr();
   Obj *real_r = GetObjPtr()->myMethod(real_p);
   return real_r->GetProxyPtr();
}
```

Figure 5: Sample proxy code for a read-only method.

Figure 5 shows the C++ stub code for a method that takes a pointer to a protected pointer as an argument and then returns a pointer to another protected object. The method `GetObjPtr` for a proxy object returns a pointer to the protected object, and the method `GetProxyPtr` for a protected object returns a pointer to the proxy for that object.

5.2 Memory management

With our scheme, memory management for a server that allows clients to have read-only access to its data segment is a little more complicated than normal. A server cannot blindly reuse the memory for a deallocated object, because there might be a client that still has a pointer stored to it. Reuse of the memory region will cause the client to read garbage data, possibly follow an invalid pointer, or even go into an infinite loop because the version numbers have been overwritten. A strategy has to be devised for reclamation of server data that prevents clients from reading invalid data, but also does not require the server to trust a client.

The strategy we have adopted is based on the waiting room strategy. A deallocated object in not put on a free-list immediately, but instead put on a waiting-list. When the server can deduce that there is no possibility that any of the clients have pointers to a certain waiting-list, the memory is recycled.

For object-based RPC the implementor can define the semantics of some operations to have the side-effect of possibly invalidating already obtained pointers. We will call such a function a *safe function*. For example, when a client disconnects, the server knows that the client has no pointers into its data. A waiting-list is annotated with the set of clients that potential have pointers to the server's data during its use. Each time a client connects, disconnects, or executes a safe function the memory manager checks if a waiting-list can be recycled.

6 Shared proxies

Read-only shared memory can not only be used to implement read-only methods, but can also be used by the object-based toolkit itself to store its meta-data. In particular, proxy objects can be located in shared read-only memory in the server. Shared proxies eliminate two problems: (i) that the server must pass protected references as capabilities to the clients, and (ii) that proxies cannot be shared between clients. Thus, using shared proxies improves both the execution and memory overhead of the toolkit.

A *shared proxy* is created by the server and located in the server's protection domain. To maintain system integrity, a client must be prevented from (un)intentionally corrupting the proxy. Therefore, a shared proxy is located in read-only memory. Pointers to shared proxies are almost completely interchangeable with pointers to local objects. Each protected object has exactly one proxy, so no pointer aliasing occurs. Also, shared proxies are callable by all clients, so proxy pointers can be stored in shared memory and exchanged between clients.

To return a reference to a protected object, the server maps the proxy read-only into the client's protection domain and returns a pointer to the proxy. The client is required to trust the server, because it cannot verify the pointer, but it avoids a relatively expensive hashtable lookup to translate a capability. A client must still pass pointers as capabilities to a server so they can be validated.

The performance increase that shared proxies provide is also achieved by eagerly attaching proxies into a client's protection domain when the client initially imports a reference to a single protected object, thereby avoiding the cost of potentially instantiating a large number of local proxies. For example, shared proxies are useful when a new domain is instantiated in Opal. The Opal system can instantiate proxies for all its system services, simply by attaching a single segment containing all the shared proxies to the new domain.

There are a few subtleties with shared proxies. A mechanism for efficiently locating RPC channels dynamically must be devised, and binding to shared proxies is different than binding to ordinary proxies. The next two subsections discuss these issues in turn.

6.1 Channel-free proxies

A proxy contains a reference to the RPC channel that it uses to communicate with the server. These channels are inherently domain specific on most operating systems. A shared proxy can be used by an arbitrary number of clients at the same time, so it has to dynamically select the channel to use on each call of a write-method.

The toolkit maintains a hashtable (the channel table) in each client that caches all the channels that connect a client to its servers. Therefore, to communicate with a server, a shared proxy must retrieve the proper channel from the channel table belonging to the client executing the call. For the toolkit, we modified our threads package so that each TCB (Thread Control Block) contains a reference to the thread's channel table; each client can then access its own channel table and lookup the channel to use.

Integrity is maintained, even though shared proxies access unprotected channel tables, because proxy code is executed by clients and a client can only access its own channel table. A corrupted table will not affect the server or other client applications. Reclamation of channels when a client terminates is trivial, because they are stored locally in the client.

The overhead of making w-method invocations on shared proxies is necessarily higher than on ordinary proxies, because a channel is looked up on each call. However, access to an object-based server is expected to often consist of following pointers from one object to another object in the same server, i.e., a client thread will with high probability use the same channel as it used on the previous call on a shared proxy. The lookup scheme has been optimized for this situation by caching the most recently used channel in a thread's TCB. The added overhead of looking up channel objects is small compared to the overhead already present in making RPC calls.

6.2 Binding

Normally, when a client binds to a protected object the server passes it a capability; the client uses the capability to instantiate a local proxy and to retrieve an RPC binding to the server. However, when a client receive a capability for a protected object with a shared proxy, a method for converting that capability to a shared proxy pointer has to be devised.

The toolkit knows statically if a capability refers to an object with a shared proxy, so it can transparently handle the binding. Each protected object has a hidden bootstrap method that maps its shared proxy read-only into the caller's protection domain and returns a pointer to the shared proxy. A capability contains enough information to both identify the server and the object, so the toolkit can invoke the bootstrap method. This method requires an RPC call, but it is an uncommon case that only occurs when a client connects to a new server, for example, by looking up its capability in a name server.

7 Evaluation

This section presents a performance evaluation of the shared memory techniques for improving the performance of object-based RPC. We have constructed a prototype implementation of the techniques for the Opal system. An existing object-based RPC toolkit was modified to support read-only methods and shared proxies.

To evaluate the effect of shared proxies and read-only methods, four different implementations of a name server and a client were built:

- **Unprotected.** The client and the server execute in the same protection domain. This version gives a lower bound for the execution times.
- **Protected.** The client and server execute in separate protection domains. Calls to the server are done with RPC calls and neither shared proxies nor read-only methods are used.
- **Shared.** The client and server execute in separate protection domains and shared proxies are used for all protected objects. Shared proxies are created lazily when a reference to an object is first exported to a client.
- **Read-only.** In addition to using shared proxies, read-only methods are also used. Shared proxies are created eagerly, so they can be accessible to the r-methods.

Table 1: Performance Measurements (in seconds)

	Unprotected	Protected	Shared	Read-only
Attach	0.00	0.13	0.13	0.13
Create	0.29	0.77	0.75	0.80
Traverse	**0.03**	**1.45**	**1.17**	**0.04**
Delete	0.18	1.74	1.45	0.59
Total	0.50	4.09	3.50	1.52

The use of shared proxies and read-only methods are completely transparent to the client and server. They are just linked with different proxy and guard stub code.

The experiments are run on a dedicated DECstation 5000 running Mach 3.0 and a prototype of the Opal server. The Opal server provides an environment in which all applications run in a single address space on top of the Mach kernel. Opal's object-based RPC toolkit uses Mach ports as its transport mechanism, and read-only sharing of memory is directly implemented as *attach-segment* calls to the Opal server.

The results of the experiments are shown in Table 1. The table shows the absolute times for coordinating sharing of memory (*attach*), creating a directory tree, traversing it, deleting it, and the total execution time. The directory tree had a total of 3069 objects.

The two first columns shows the effect of inserting a protection barrier between the client and the name server. It makes the execution-time of *Traverse*, which consist of read-only calls, almost 50 times slower.

Introducing shared proxies makes creating, traversing, and deleting the tree slightly faster. For Traverse, the overhead of instantiating local proxies is eliminated. The performance is also improved because the server returns pointers directly to shared proxies, thereby eliminating hashtable lookups to convert capabilities into local proxy pointers. The results show that the extra cost of retrieving the channel from a thread's control block is dominated by the savings from passing proxy-pointers directly and eliminating instantiation of local proxies.

Read-only methods significantly improves the performance of traversing and deleting the structure. Creating the structure is slightly slower because shared proxies and guards are created eagerly. Traversing the structure in the Read-only version is a little slower compared to the Unprotected version, because the stub code for the read-only methods must convert pointers and call the method indirectly.

The total execution times show that the extra time spent attaching segments read-only to clients can easily be amortized by the use of shared proxies and read-only methods, thereby improving the overall performance of object-based RPC.

8 Conclusion

This paper presented a technique for using read-only shared memory to improving the performance of method invocation between a server and untrusted clients. Using the technique, queries to protected objects execute within the client's domain; queries that modify server data use conventional cross-domain calls. Version-based synchronization is introduces to coordinate object access.

The technique is applicable when data is read-mostly and protected updates must be enforced. The server gains the speed-up, by trading away fine-grained read-access control to its objects. The technique gives the programmer some of the benefits that a safe language provides without having to pay the full price, i.e., trusting each other and the compiler.

Our experiments have shown the potential performance benefit of read-only shared memory and version-based synchronization. For example, with our benchmark, read-only methods execute a read-only traverse transaction nearly as fast as an unprotected local call, but with the protection (for the server) as great as in the fully protected case; in contrast, a fully-protected implementation sacrifices almost an order of magnitude in performance.

Acknowledgements

We would like to thank Mike Feeley and Ashutosh Tiwary for helpful comments and assistance with the Opal implementation.

References

[1] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[2] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May 1986.

[3] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1), February 1990.

[4] Ellis E. Chang and Randy H. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented DBMS. In *ACM International Conference on Management of Data*, 1989.

[5] K. Grønbæk and R.H. Trigg. Design Issues for a Dexter-based Hypermedia System. *Communications of the ACM*, 37(2):40 – 49, February 1994.

[6] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1992.

[7] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. *ACM Trans. on Computer Systems*, 12(4), November 1994.

[8] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[9] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[10] J.E. Allchin and M.S. McKendry. Synchronization and recovery of actions. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 31–44, August 1983.

[11] Amitabh Dave, Mohlalefi Sefika, and Roy H. Campbell. Proxies, application interfaces, and distributed systems. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, September 1992.

[12] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the 1993 Summer Usenix conference*, June 1993.

[13] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network Objects. In *Proceedings of the 14th ACM Symp. on Operating Systems Principles*, pages 217–230, December 1993.

[14] S. Brandt and O.L. Madsen. Object-Oriented Distributed Programming in BETA. In R. Guerraoui, O.M. Nierstrasz, and M. Riveill, editors, *Lecture Notes in Computer Science 791*, pages 185 – 212. Springer-Verlag, 1994.

[15] John Rosenberg. Architectural and operating system support for orthogonal persistence. *Computing Systems*, 5(3), July 1992.

[16] Maurice Herlihy. A methodology for implementing highly concurrent data objects. Technical Report CRL 91/10, DEC Cambridge Research Laboratory, 1991.

[17] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report 92-03-02, Columbia University, Department of Computer Science, June 1991.

[18] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[19] M. Anderson, R. D. Pose, and C. S. Wallace. The password-capability system. *The Computer Journal*, 29(1):1–8, February 1986.