TECHNICAL REPORT 98-03

Integrating Remote Procedure Calls with Virtual Memory Management for Cooperative Autonomous Systems

Kenji Kono, Kazuhiko Kato, and Takashi Masuda

September 14

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TITLE
Integrating Remote Procedure Calls with Virtual Memory Management for Cooperative Autonomous Systems

AUTHORS
Kenji Kono, Kazuhiko Kato, and Takashi Masuda

KEY WORDS AND PHRASES
remote procedure call, remote method invocation, distributed shared memory, distributed object, distributed operating system

ABSTRACT
Cooperation and autonomy of distributed processes are indispensable features of loosely coupled applications where information is shared beyond administrative boundaries such as laboratories and companies. To facilitate this style of information sharing, this paper proposes a distribution facility, called the VM-integrated RPC, that integrates the remote procedure call (RPC) with virtual memory (VM) management. It combines the advantages of the conventional technologies for distributed programming: RPC and distributed shared memory (DSM). By incorporating data-sharing semantics, the VM-integrated RPC enables effective cooperation of processes like DSM technologies do. By retaining the advantages of RPC, it preserves the autonomy of processes and is also applicable to heterogeneous environments. The VM-integrated RPC shows reasonable performance that is scalable to the access ratio of remote shared data. In addition, it adaptively changes caching strategies and coherency protocols for better performance across a wide range of access patterns.

REPORT DATE	**WRITTEN LANGUAGE**
September 14	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
29	41

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT
Submitted for publication

DISTRIBUTION STATEMENT
This technical report is available ONLY via anonymous FTP from `ftp.is.s.u-tokyo.ac.jp` (directory `/pub/tech-reports`).

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Integrating Remote Procedure Calls with Virtual Memory Management for Cooperative Autonomous Systems

Kenji Kono† Kazuhiko Kato‡ Takashi Masuda†

† Department of Information Science,
Graduate School of Science,
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113–0033 Japan
Email:{kono,masuda}@is.s.u-tokyo.ac.jp

‡ Institute of Information Sciences and Electronics,
University of Tsukuba
Tsukuba, Ibaraki 305–8573 Japan
Email: kato@is.tsukuba.ac.jp

1 Introduction

The widespread use of computers and networks by the general public has stimulated the sharing of information across the administrative boundaries of organizations such as laboratories, departments, and companies. One of the distinguished features of such information sharing is that the *cooperating* processes that share information are highly *autonomous*, since they represent organizations independent of one another. Cooperating processes exchange and share information for a certain period of time, and during this time all they know about each other is the logical interfaces they provide. Each process enforces its own policy with regard to the management and protection of its local resources. We refer to distributed systems intended to facilitate this style of information sharing as *cooperative autonomous* systems [11].

A wide variety of distributed applications require both autonomy and cooperation. When two companies trade in some merchandise, for example, they are required to cooperate with each other on the trade. But because they are socially independent of each other, their autonomy must be respected even during the trade; cooperating processes do not expose their internals to each other. Another example is a "collaboratory" [27], which helps researchers cooperate on a particular research subject. Each laboratory is autonomous in that it usually carries out research independently of the other laboratories, but sometimes (when necessary) each laboratory cooperates in sharing research results. These autonomous cooperative systems are one of the target applications of contemporary standardizations such as OSF/DCE [33], OMG/CORBA [31], and DCOM [10].

1.1 Remote Procedure Call and Distributed Shared Memory

Two technologies for distributed programming have been widely used: the remote procedure call (RPC) [40, 5, 4, 16, 19, 14, 2, 36] and distributed shared memory (DSM) [28, 37, 30]. There are three distinctive differences between these technologies [11]. One is that the RPC inherently provides the data-passing semantics, whereas DSM provides the data-sharing semantics. The second is that the RPC provides distribution transparency by means of a high-level language abstraction of parameter passing, whereas DSM provides transparency by simulating a logical shared address space over a set of physically distributed local memory systems. And the third is that an RPC requires agreement only on logical interfaces, whereas DSM requires agreement on physical memory layouts.

The sharing nature of DSM eliminates the diversity of address spaces, thereby enabling any type of data including address references (pointers) to be shared by communicating processes. This property of DSM allows processes to cooperate closely, but it precludes autonomy of the cooperating processes. Since all the data in a shared address space is exposed to all the processes, the processes must agree on physical memory layouts *a priori* and that all the memory contents are open to all the other processes. DSM is therefore especially suitable for tightly coupled applications, such as parallel numeric computations.

The use of RPCs, on the other hand, does not compromise the autonomy of communicating processes because each process has only to agree on logical interfaces and can use its own management policy.

Applications in loosely coupled distributed environments — such as mail, news, and the WWW — are therefore typically based on RPC technology. Data-sharing semantics, however, is not easily provided by an RPC technology because address references are valid only within a single address space and cannot be passed beyond address space boundaries. Consequently, cooperating processes cannot share data by simply transmitting remote references; to implement data-sharing semantics, application programmers must design a protocol for each application. This drawback of RPC technology is well known and pointed out in the literature [3].

1.2 The Approach: VM-Integrated RPC

This paper proposes a novel scheme developed to support cooperative autonomous systems effectively. The novelty of the scheme is that it combines the advantages of RPC and of DSM to allow cooperation and autonomy at the same time. Since autonomy is indispensable in loosely coupled distributed systems, we take an approach to extend RPC technologies for cooperation. The proposed scheme, the *VM-integrated RPC*, integrates RPC with virtual memory management. Like conventional RPC technologies, it preserves autonomy because cooperating processes share only logical interfaces. The scheme nevertheless enables communicating processes to pass any type of data including address references beyond address space boundaries. This feature of the scheme facilitates process cooperation like DSM technologies do.

We have already reported preliminary version of this scheme [24] and have applied it to object migration [25]. The present paper describes the scheme comprehensively and reports a prototype implementation called TRAP. It also shows the results of measuring the performance of the scheme. In communication-bound benchmarks, the proposed scheme shows performance competitive with the optimal case in which all access patterns are assumed to be known statically. Even in the worst case, our scheme increases the execution time by approximately 20% compared with that optimal performance. In computation-bound benchmarks, the proposed scheme performs 20% to 30% better than schemes that do not utilize a mechanism for virtual memories.

The central technology of the VM-integrated RPC combines three techniques into one sophisticated RPC mechanism: (1) exploitation of general hardware for virtual memory management, (2) coherency protocols, and (3) RPC-based stub generation (i.e. language preprocessing). Generally, a reference across an address space boundary is meaningless unless dereference operations are executed at the address space where it is created. Conventional RPC systems [40, 5] and object-oriented distributed systems [22, 34, 6, 18, 21, 8] *callbacks* the target address space every time a remote reference is dereferenced. The VM-integrated RPC automatically detects the first dereference of a remote reference by using a memory management unit (MMU), which is equipped with virtually all computers from personal ones to mainframes. Thereafter, the referenced data is locally *cached* so that the subsequent accesses incur no more overhead than do local memory accesses.

This portion of our system was inspired by the "pointer swizzling at page fault time" proposed by Paul Wilson [38, 39], a method that efficiently extends hardware-provided address space. The VM-integrated RPC is not, however, intended to provide a huge address space but rather to address the problems caused by the cooperation and autonomy of distributed processes.

Our technique is advantageous with regard to both transparency and efficiency. It provides transparency at the instruction-code level. That is, the instruction codes that manipulate local references can without any modifications manipulate remote references. Furthermore, the technique is efficient because communication overhead is reduced as a result of caching.

To allow update operations on cached data, the scheme includes the protocol that guarantees the coherency of original and cached data. In addition, the scheme controls concurrency properly to preclude multiple processes from being falsely interleaved and causing an incorrect status. Extensive exploitation of the synchronous property of RPC makes these protocols simple and light. Like a conventional RPC system, a language preprocessor is required to generate *stubs* that conceal communication details although the runtime mechanism is essentially language-neutral. Since our language preprocessor generates stubs that marshal and unmarshal[1] data structures according to target architectures, cooperative processes have to agree only on logical interfaces. Thus our system is applicable to heterogeneous environments.

[1] In Java/RMI [21], marshal/unmarshal is called serialize/deserialize.

1.3 Organization of the Paper

This paper is organized as follows. Section 2 clarifies the implementation of transparent remote references in RPCs. Section 3 defines the execution model of TRAP and defines the semantics of remote references. Section 4 describes the mechanism of the VM-integrated RPC. Section 5 presents the coherency protocols. Section 6 describes our programming language preprocessor. Section 7 reports experimental results, and Section 8 concludes the paper.

2 Remote References in Conventional RPCs

To preserve the autonomy of processes, conventional RPCs do not share a single memory image among cooperating processes but instead reconstruct data structures at each address space. As pointed out in Silva et al. [12], however, owing to the lack of a logical shared address space, most RPC systems compromise on the semantics of remote references. In conventional RPCs, when a reference is transmitted to a remote address space, a *copy* of the referenced data is created; that is, the referenced data is passed by *value*. The identity of the created copy is generally different from the identity of the original data and the correspondence between them is not preserved. To obtain the *data-sharing* semantics, application programmers must write code to explicitly write back modifications to original data. This burden on programmers is lighter when the system automatically guarantees the data-sharing semantics; a "transparent remote reference" should preserve the identity of the remotely referenced data across address space boundaries. To explain the issues in implementing remote references, this section reviews conventional schemes to implement transparent remote references. Section 2.1 clarifies the criteria for comparing implementation schemes. Section 2.2 reviews those schemes, and Section 2.3 compares them according to the criteria.

2.1 Criteria for Schemes Implementing Remote References

Advanced distributed systems and languages provide application programmers with a view that remote references are, at least at the language level, indistinguishable from local references. This implies that some checks must be performed at runtime if remote references are to be distinguished from local ones. These checks are often called *presence tests* and the overhead due to presence tests is the cost of transparency. One of the criteria is thus whether the introduction of remote references degrades performance of the other parts of the entire system. Someone who never uses remote references should not pay the costs involved in the implementation of remote references.

Another factor to consider is the cost of network communications, and there are two criteria to estimate the cost of network communications. First, the *number* of network communications should be reduced. Second, the *amount* of network data transfer should be reduced. These two criteria frequently contradict each other. Consider an example in which a reference to the root of a binary tree is remotely transferred to a callee process. To reduce the *number* of network communications, it is best to transmit the entire tree *at once*. To reduce the *amount* of network data transfer, it is best to transmit tree nodes on demand that the callee actually attempts to visit. Since it is generally quite difficult to predict the access pattern of the callee, it is almost impossible to reduce both the number of communications and the amount of data transfer at the same time by transmitting *at once* the nodes that the callee *actually* visits.

To summarize, there are three criteria for execution performance of the implementations of remote references: (1) the cost payed to presence tests, (2) the number of network communications, and (3) the amount of network data transfer. The first criterion affects local computation traversing a mixture of local and remote references, and thus dominates computation-bound applications. The last two criteria affect the cost of network communication, and thus dominate communication-bound applications.

2.2 Implementation Schemes

In many *non-distributed* systems and languages, a local reference is implemented as a virtual address because a processor can interpret virtual addresses directly, without software intervention. Since a virtual address is invalid outside the address space where it was created, transmitting it to a remote address space is meaningless. Instead, we can transmit a *logical identifier* for accessing the data referred to by the virtual address. Receiving the logical identifier, the receiver remotely operates on the referenced data through the identifier. Whenever a remote operation is requested, the runtime system *callbacks* the sender to perform the requested operation. We call this the *callback* scheme. It is widely used in distributed object-oriented systems [22, 34, 6, 8, 21, 31] in which an object identifier (OID) directly acts as a logical identifier. In the callback scheme, presence tests are required to distinguish logical identifiers from virtual addresses.

The callback scheme not only incurs the overhead of presence tests, but also greatly increases the number of network communications because a network communication occurs every time a remote reference is dereferenced. An appealing approach to reducing the number of callbacks is to *cache* the remote data. Once it is cached, it can be accessed without performing additional communications. This is attractive since it reduces both the number of network communications and the amount of data transferred. With respect to the timing of cache creation, the caching schemes can be classified into two types: the *lazy caching* schemes and the *eager caching* schemes.

A lazy caching scheme creates a cache *lazily*, or *on demand*. That is, the creation of a cache is deferred until an attempt to access remote data is made. Since remote data are cached one by one, some literature refers to this type of scheme as a *shallow* scheme. To distinguish references to cached data from references to remote (not cached yet) data, this scheme requires presence tests. The eager caching scheme, on the other hand, caches the transitive closure of a remote reference *at once*, or *eagerly*. Since the copy of the entire closure is created at once, this kind of scheme is sometimes called a *deep* scheme. An eager caching scheme does not require presence tests because there are no references across address space boundaries; all the remote data are always cached locally. In both kinds of schemes, the runtime systems must manage the coherency and identity issues arising from creation of multiple copies of the same data. If the cost of the coherency maintenance is terribly high, these caching schemes become unattractive in the cases where update operations are frequent. Table 1 summarizes the three schemes described above. Distributed C [23] provides a language construct to choose between the callback scheme and the eager caching scheme.

scheme	description
callback	Whenever operating on a remote data, the requested operation is delegated to a target remote address space.
lazy caching	At the first access to a remote data, the accessed data is cached locally. In other words, remote data are cached *lazily*.
eager caching	When transmitting a reference to a remote site, the transitive closure of the reference is transferred *eagerly* to the remote site.

Table 1: Schemes for implementation of remote references.

As mentioned above, the callback scheme and the lazy caching scheme require presence tests to distinguish remote references from local references. Several techniques for performing presence tests have been already proposed (see [26]). One approach is to *tag* a reference. An untagged reference directly refers to a local data, but a tagged reference is considered to refer to a remote data. Every time a reference is dereferenced, the tag of the reference is checked. In this approach, a compiler determines the locations to insert codes for tag checks by flow-analysis; otherwise, all memory accesses need to be checked. Thus a compiler must be altered to do this insertion, and all the libraries and executables must be re-compiled by the altered compiler.

Tagged references incur the cost of presence tests even if all the data reside in a local address space. Another approach that eliminates presence tests for local data, an approach we call the *proxy-overlay* [7], has been proposed in the context of distributed object-orientation. In the proxy-overlay, a proxy called "O-proxy" represents a remote object and is the same size as the remote object. The O-proxy's methods are bound to a code that notifies the runtime system of the first attempt to invoke one of the methods of the O-proxy. When an attempt to invoke a method of the O-proxy is made, the runtime system fetches the target remote object and overlays the O-proxy with the fetched object. The methods of the O-proxy are bound to those of the target object. Consequently, no presence tests are required after an O-proxy is overlaid with a real object.

The proxy-overlay relies heavily on language implementations. It assumes that all accesses to an object are strictly method invocations even at the instruction-code level. Otherwise, we cannot catch an attempt to access a remote object. This restricts compiler optimizations; for example, a method code cannot be inline-expanded. Consequently, the proxy-overlay is applicable to "pure" object-oriented languages but not to C++ and Java[2] because they allow direct accesses to instance variables.

2.3 Discussion

The three implementation schemes described in Section 2.2 are not satisfactory with regard to at least the following three points. First, the callback and lazy caching schemes require presence tests which usually degrade performance. Second, the lazy and eager caching schemes require a coherency protocol to guarantee the data-sharing semantics. Many systems using these schemes are reported, but they put the problem of coherency into the hands of application programmers. Third, none of these three schemes is applicable by itself to various access patterns of applications. If only a small portion of a huge data structure is remotely accessed, the callback and lazy caching schemes perform better because of the reduced amount of network data transfer. But if a large portion of the huge data structure is remotely accessed, the eager caching scheme is better because of the reduced number of network communications. Table 2 summarizes the properties of the three schemes described in Section 2.2.

scheme	presence test	coherency protocol	amount of data transfer	number of communication
callback	*R*	*NR*	○	×
lazy caching	*R*	*R*	○	×
eager caching	*NR*	*R*	×	○

R: *required*
NR: *not required*
○: *relatively small*
×: *relatively large*

Table 2: Comparison of schemes for implementation of remote references.

The VM-integrated RPC proposed in this paper overcomes the three problems described above by combining the advantages of the three schemes. First, by exploiting MMU hardware, the VM-integrated RPC eliminates the cost of presence tests for cached data; the MMU hardware automatically performs presence tests without software intervention. If an attempt to access remote data is made, the MMU hardware detects it and notifies the runtime system. Thereafter the runtime system caches the accessed remote data properly. The cost of handling page faults is amortized over the number of accesses to cached data. The VM-integrated RPC provides distribution transparency at the instruction-code level: instruction codes that manipulate local memories can without modification manipulate remote data. This property of the VM-integrated RPC enhances reusabilities of existing libraries and compilers for distributed computing.

Second, the problem of coherency is resolved by the runtime system. Application programmers are given the simple view that the identities of remote data are preserved across address space boundaries and update operations on remote data are directly shared by other processes. Generally, coherency maintenance is difficult in environments where multiple activities run in parallel. To simplify the problem of coherency, we have tailored the execution model of our system to cooperative autonomous systems. As described in Section 3, our execution model prohibits the abuse of remote references and provides simple and clear semantics of remote references by exploiting synchronous property of RPC. Owing to the clear semantics of references, our system simplifies coherency protocols to avoid runtime overheads.

Third, in order to adaptively control the timing of caching, the VM-integrated RPC uses the scheme we call *adaptive caching*. This scheme is, in a sense, the combination of the lazy and eager caching

[2] If Java were statically compiled into native-code, the proxy-overlay would be hard to apply. In the current Java virtual machine, instance variables are accessed indirectly.

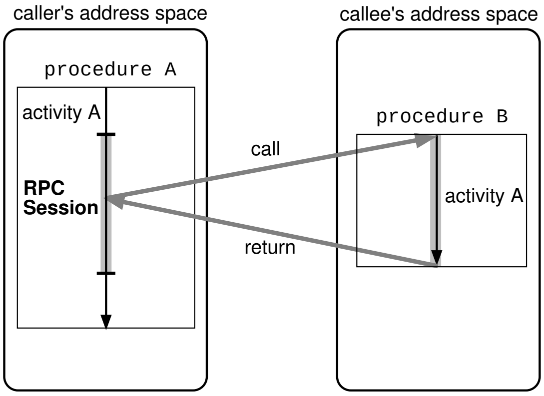


Figure 1: TRAP execution model based on RPC session. The activity A declares an RPC session at the caller, migrates to the callee, and executes a remote procedure.

schemes. The VM-integrated RPC caches only a *subset* of a transitive closure of a transmitted reference, not the entire closure. Application programmers can dynamically alter, according to the access pattern of a called remote procedure, the size and the "shape" of a subset to be cached at once. If the entire closure is always transferred, the runtime behavior of the VM-integrated RPC resembles that of the eager caching scheme. If remote data is transferred one by one, it resembles the lazy caching scheme. As our experimental results in Section 7 show, the proper customization of an algorithm to specify a transferred subset of a closure greatly improves performance.

3 RPC Session Model and Its Sharing Semantics

To describe the VM-integrated RPC rigorously, this section defines the execution model of the VM-integrated RPC and introduces some basic terminology. Section 3.1 presents our execution model called RPC session. Section 3.2 defines the sharing semantics of RPC sessions.

3.1 RPC Session Model

In the conventional RPC model of execution, a distributed thread of control, called an *activity*, executes program code, migrating from one address space to another. When an activity invokes RPC, it migrates from the caller's address space to the callee's and executes the invoked remote procedure. After the execution of the procedure, the activity returns to the caller's address space and continues execution. An activity can execute nested invocations of RPCs. That is, the activity can invoke another RPC before finishing the outstanding RPC. This execution model implies that RPC is synchronous. That is, it implies that no activity remains at the caller's address space when a remote procedure is executed. In this conventional model, we can obtain concurrency by explicitly spawning new activities. Figure 1 illustrates an invocation of RPC by an activity. When the activity A invokes RPC, it migrates from the caller's address space to the callee's and executes the invoked procedure. After the execution, the activity A returns to the caller.

The abuse of remote references causes undesirable situations. First, from a semantic point of view, the abuse of remote references vitiates the autonomy of cooperating processes because a reference to a remote address space implies semantic dependence on that remote address space. When processes finish cooperating, remote references between them must be disposed of explicitly. Otherwise, their dependence continues and they lose their autonomy. Second, from a viewpoint of performance, the abuse of remote references greatly increases the cost of the data-sharing semantics. For the data-sharing semantics, the runtime system pursues the outflow of references, and keeps track of the one-to-one correspondence between a reference and the referenced data. The cost of keeping track of the correspondence would be large if remote references were spread widely over a lot of address spaces, in the worst case, over

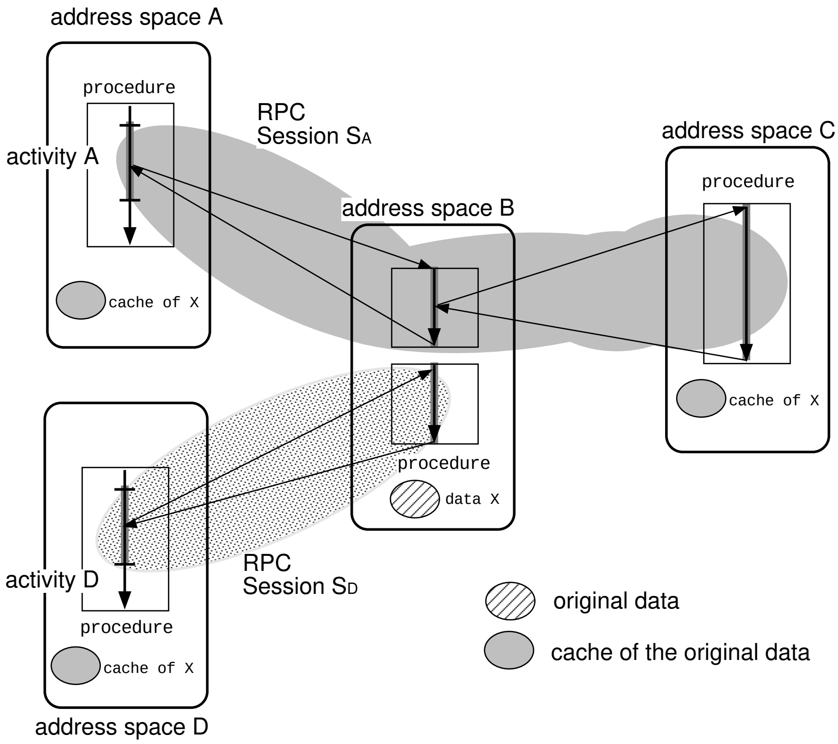


Figure 2: Multiple RPC sessions executing concurrently. RPC session S_A executes two RPCs in a nested fashion. If shared data are updated by the two sessions in an interleaved way, the inter-session protocol serializes the updates while the intra-session protocol makes the caches coherent.

worldwide networks.

These problems are due to application programmers not having a way to notify the runtime system that some remote references are no longer needed; there are no primitives to control the *term of validity* of remote references. Our execution model provides a primitive that declares *RPC session*, which controls the term of validity. RPC session imposes two constraints on the use of remote references: *temporal* and *spatial* constraints. The temporal constraint determines the period during which remote references are allowed to be used. Within RPC session, application programmers can transmit references beyond address space boundaries, and the data-sharing semantics is guaranteed by the system. Outside RPC session, on the other hand, no references are allowed to refer to remote address spaces. If a remote reference is dereferenced outside RPC session, the runtime system raises a runtime exception.

The spatial constraint determines a set of address spaces to which remote references are allowed to refer. During execution of RPC session, remote references are allowed to designate only the address spaces that *participate* in the executing RPC session. When an activity executing RPC session invokes a remote procedure, the address space of the invoked remote procedure becomes a participant in the RPC session. If an attempt to pass a reference to an address space outside RPC session is made, the runtime system raises an exception to notify the violation of the spatial constraint. Owing to the temporal and spatial constraints imposed by RPC sessions, remote references are *enclosed* within RPC sessions.

In Fig. 2 two RPC sessions S_A and S_D are executing concurrently. S_A performs two RPCs in a nested fashion. The participants of the session S_A are the address spaces A, B, and C; and the participants of S_D are the address spaces B and D. According to the spatial constraint on the use of remote references, the session S_A cannot contain references to the address space D because the address space D is not a participant of the session S_A. Similarly, the session S_D cannot contain references to the address spaces A and C because they do not participate in S_D. When the session S_D is running within the address space B, it may attempt to transmit a reference designating the address space A to the address space D, violating the spatial constraint. In this case, the runtime system on the address space B raises an exception.

Owning to the synchronous nature of RPC, each RPC session contains only one activity. Although concurrency within an RPC session is not allowed, we can obtain concurrency by executing multiple RPC sessions simultaneously. We spawn some activities ahead of time and have each activity initiate RPC session.

3.2 Data-Sharing Semantics

In our execution model, a data of any type can be *shared* by any activities that possess references to the data. If some data is required to be shared by some address spaces, it is necessary only to duplicate a reference to the data and pass it to other address spaces. Although multiple activities run concurrently and migrate from one address space to another, the system guarantees the *data-sharing* semantics, in which a dereference operation on a reference always returns the value assigned by *the most recent* update on the reference. As known well, the lack of a consistent global clock in a distributed system makes it difficult to determine which update is the most recent.

To solve this problem, we extensively exploit the properties of RPC session. Since remote references are effective only within RPC session, the operations on shared data are all performed within RPC session. If the effect of executing RPC sessions in an interleaved fashion is equivalent to executing them *serially* in some order, we can decide the order of dereferences and assignments on remote references. As described in Section 5.1, the *inter-session* protocol guarantees the serializability of RPC sessions. Within each RPC session, the *intra-session* protocol keeps track of the correspondence between a reference and the up-to-date value designated by the reference and guarantees the data-sharing semantics within the RPC session.

In Fig. 2, if the data X on the address space B is updated by two RPC sessions in an interleaved fashion, the inter-session protocol serializes the updates and schedules the two RPC sessions S_A and S_D either in the order S_A, S_D or in the order S_D, S_A. Within each RPC session, the coherency of caches is maintained. The data X instantiated at the address space B is cached to the other address spaces. The three caches on the participants of S_A (the address spaces A, B, and C) are made coherent by the intra-session protocol, when the RPC session S_A is running, whereas the two caches on the participants of S_D (B and D) are made coherent when S_D is running.

4 Basic Mechanism of the VM-Integrated RPC

This section describes the mechanism of the VM-integrated RPC, which is an efficient implementation of RPC session and its sharing semantics described in Section 3. As mentioned in Section 2.3, the VM-integrated RPC has two advantages over the conventional RPCs. One is that it eliminates presence tests by extensively using Paul Wilson's "pointer swizzling at page fault time" technique [38, 39]. And the other is that it uses the adaptive caching scheme that combines the merits of both the lazy and eager caching schemes. Section 4.1 describes the basic mechanism of the VM-integrated RPC. Section 4.2 presents the treatment of aliasing and cyclic data structures. Section 4.3 explains the way the adaptive caching scheme is incorporated into the VM-integrated RPC, and Section 4.4 describes some properties of the VM-integrated RPC.

4.1 Basic Mechanism

For transparency at the instruction-code level, Trap represents even a remote reference as an ordinary virtual address. Since a virtual address becomes meaningless outside an address space, before a reference is transferred it must be converted into a *universal ID* (UID) valid in the entire distributed system. A UID comprises three components: an address space ID, an address valid in that address space, and a type ID of the referenced data. An address space ID indicates the *birthplace* of the referenced data. The actual implementation of an address space ID depends on the transport layer of the underlying communication system. Our implementation built on Unix and TCP/IP uses a pair consisting of an IP address and a port number. As described later in this subsection 4.1, a type ID is required to preserve data types in heterogeneous environments. We assume that the system can obtain the type ID of an actual data structure from a Trap name server. The conversion from a UID to a virtual address is called

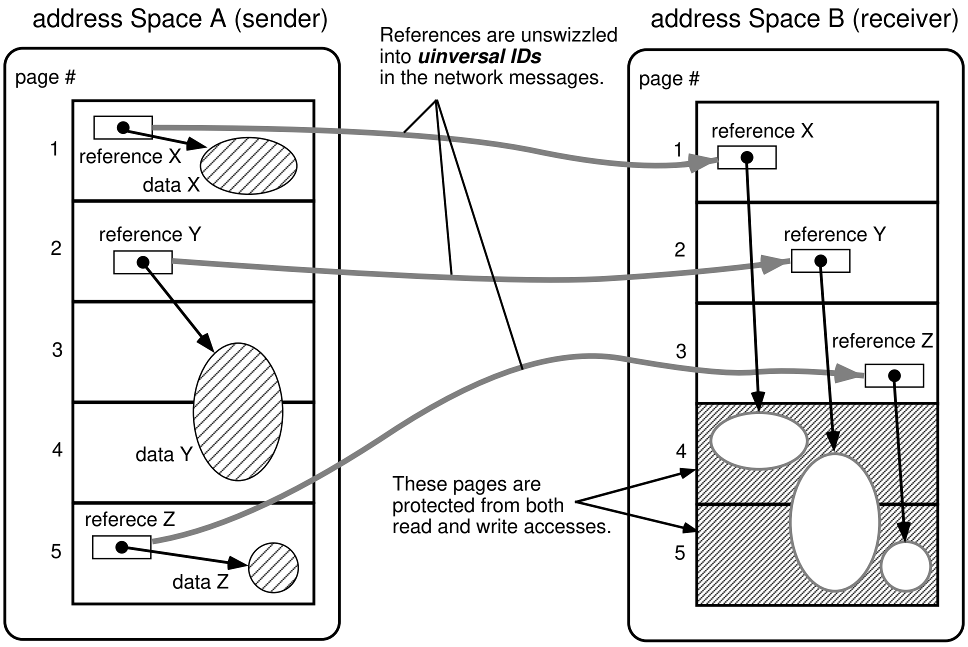


Figure 3: Just after the three references X, Y, and Z are swizzled at the receiver. The references are transferred in the form of UIDs. The receiver swizzles the transferred UIDs by allocating empty protected areas.

swizzling, and the conversion of a virtual address to a UID is called *unswizzling*. These terminologies are borrowed from persistent object systems and persistent languages [29, 9, 35].

When a reference is passed as RPC argument or returned as a result, the reference is transmitted from one address space to another. On the transfer of a reference, it is unswizzled into a corresponding UID so as to identify the referenced data in the entire system. UIDs can be regarded as a canonical representation of virtual addresses and appear only in network messages. Receiving a UID, the receiver tries to swizzle the received UID. But how does the receiver swizzle the UID into a virtual address? No virtual addresses at the receiver can refer to the referenced data directly at this point because the referenced data resides only in the sender's address space.

To resolve the issue, we apply Wilson's "pointer swizzling at page fault time" technique. The runtime system allocates a *protected* area in a memory page (or pages) which the MMU protects from read and write accesses. The size of the allocated area is the same as that of the referenced data, whose size can be obtained from the type ID embedded in the received UID. The UID is swizzled into the address of this allocated area. Note that this area is *empty* at this point; the actual data *will* be copied to the location later when necessary. Figure 3 illustrates this situation. Three references X, Y, and Z are transferred from the sender to the receiver. At the receiver three empty areas are allocated respectively for each data in two protected pages. As shown in the figure, one protected page can contain multiple remote data, and a data can cross page boundaries.

The first attempt to access data allocated in a protected page is detected by the MMU and at that time an access-violation exception is raised. The operating system kernel is informed *a priori* that the runtime system handles this exception. This functionality is usually supported in many modern operating systems such as Mach, Linux, FreeBSD, and Solaris. Catching this exception, the runtime system determines the location at which the exception was raised, fetches the accessed data, and releases the access protection of the page. At this time, only the read access is permitted to detect a write access afterwards. Then the computation proceeds normally. Note that all the data allocated in the same page must be fetched because the subsequent accesses to the page cannot be detected after the page protection is released. To fetch all the data within a page, we need a UID to have a type ID. If those data refer to one another, we can determine all data types from the type of a root data. In our system, however, since some data in a page may have nothing to do with each other, we may be unable to determine their type IDs. In Fig. 4, the MMU hardware detects the first attempt to access page #5. Then the data Y

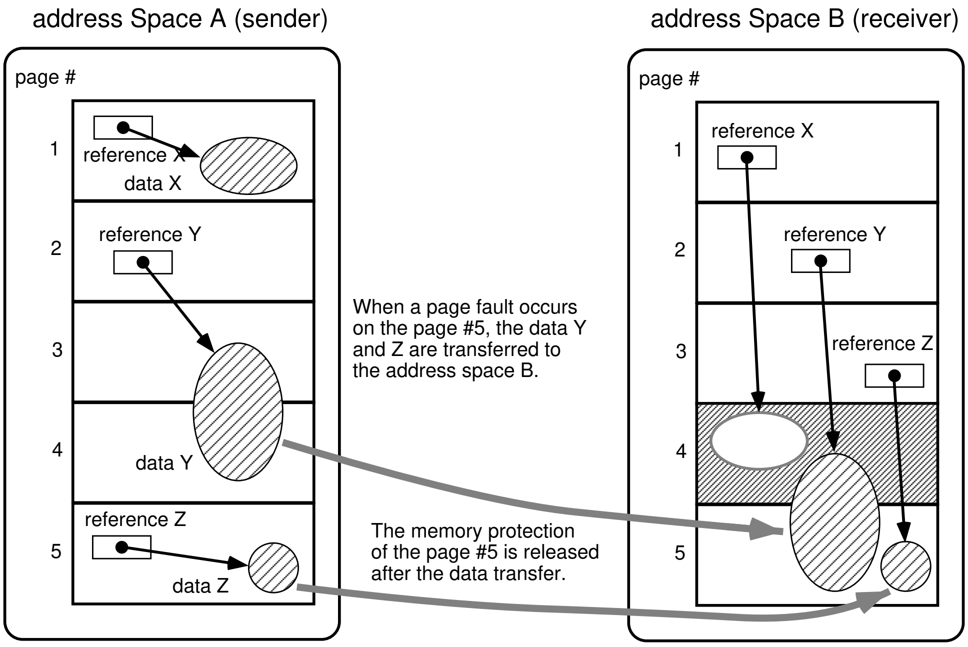


Figure 4: Just after the detection of the first attempt to access the data Z. The MMU hardware detects the attempt. Then the data Z and data Y are transferred to the receiver.

and Z, which reside on page #5, are fetched from the sender's address space.

4.2 Treatment of Aliasing and Cyclic Structures

The mechanism described above works well even if two references by chance designate the same data. The TRAP runtime system manages all the data that may be referenced from remote address spaces. The runtime system maintains a *data allocation table (DAT)* which records some attributes of every data under TRAP's control. Each entry of a table consists of four elements: a page number on which the data is allocated, an offset within that page, a UID of the data, and a *hot location*. A hot location is an address space from which the up-to-date data is available. The details of a hot location are described in Section 5.2. In the example shown in Fig. 4, the receiver's DAT would be like Table 3. When a reference to the data Z is dereferenced at the receiver, the runtime system looks up its DAT and infers that the data Y crosses a page boundary because the offset_Y plus the size of the data Y is greater than the page size. Then the data Y and data Z are fetched from the sender's address space.

page #	offset	UID	Hot location
4	offset_X	X	A
4	offset_Y	Y	A
5	offset_Z	Z	A

Table 3: Data allocation table at the address space B (receiver).

By looking up a DAT, the runtime system handles aliases and cyclic data structures correctly. When a receiver swizzles a UID, it searches for a corresponding entry in the receiver's DAT. If the entry is found, the UID is swizzled into the corresponding virtual address. If not found, the technique described in Section 4.1 is used to swizzle the UID. When two references refer to the same data, their UIDs are identical, so we surely swizzle them into the same virtual address. Consequently, our system maintains aliases and cycles to preserve logical relations between data across the boundaries of address spaces.

4.3 Incorporation of the Adaptive Caching Scheme

When transmitting a reference or fetching a remote data, the TRAP runtime system transfers a set of data: a subset of the transitive closure of the transmitted reference or data. There are many algorithms that take a subset of a transitive closure, and by default TRAP takes a certain depth of a transitive closure in the breadth-first manner. Application programmers can change the depth by specifying a *closure size* parameter that bounds the amount of a transferred closure. The lazy caching scheme is equivalent to setting the closure size parameter to zero, and the eager caching scheme is equivalent to setting the parameter to infinity. Furthermore, application programmers can replace the default algorithm with their own algorithms for each of the data types.

The routines that marshal and unmarshal a closure (which is called *pickling*) according to logical data types are generated by a language preprocessor. The details of these routines are described in Section 6.3. On the transfer of a data, the internal representations of the data are encoded and decoded to preserve their types in heterogeneous environments. Except when encoding and decoding references, we can use the well-known methods. References embedded in the transferred data, however, must be unswizzled (encoded) and swizzled (decoded).

4.4 Discussion

The overhead of pickling is often pointed out as the performance bottleneck of RPC. This overhead, however, is a necessary cost of preserving the autonomy of address spaces. If the autonomy is not needed and hardware homogeneity can be assumed, we can eliminate this overhead by transferring a physical image of a virtual memory page without conversions. This is the approach most DSM systems take. From the standpoint of the VM-integrated RPC, DSM systems are optimized by assuming hardware homogeneity and so on. From the standpoint of DSM systems, our VM-integrated RPC makes fewer assumptions, but does so at some cost.

Although most DSM systems assume hardware homogeneity, there have been some research devoted to the development of a *heterogeneous* DSM that allows processor architectures to be heterogeneous. The DSM's nature of sharing a single address space requires every data component to be aligned at the same address on all the heterogeneous machines, but heterogeneous processors enforce different alignment requirements. To overcome this problem, a heterogeneous DSM called Mermaid [41] inserts dummy elements so as to force each data component to be aligned at the same address on all the machines. All the compilers must therefore be altered to use the same data layout, and all applications must be recompiled by an altered compiler. Moreover, these procedures must be repeated every time either a new compiler or a machine with a new architecture is added to the system. Consequently, as pointed out by Stumm and Zhou [37], heterogeneous DSMs do not scale to actual heterogeneous environments.

5 Implementation of Data-Sharing Semantics

Our semantics of remote references described in Section 3.2 requires the runtime system to serialize multiple RPC sessions. In Section 5.1, we describe the *inter-session* protocol that does this. Since our mechanism creates multiple caches of the same data, the runtime system also maintains the coherency between the original data and its caches. Section 5.2 describes the *intra-session* protocol maintaining the coherency of multiple caches.

5.1 Inter-Session Protocol for Serialization

To serialize multiple RPC sessions, we can use protocols developed for distributed transactions. Many algorithms for implementing distributed transactions have been already proposed and it is inherently possible to incorporate any of those algorithms into the VM-integrated RPC. Which protocol is appropriate, however, depends on the characteristics of the applications. In cooperative autonomous systems, each activity is usually running independently of the others and accesses to shared data are restricted to the period during which processes cooperate with each other. Thus it is expected that there is little interference between RPC sessions. We therefore use an optimistic protocol to serialize RPC sessions. We refer

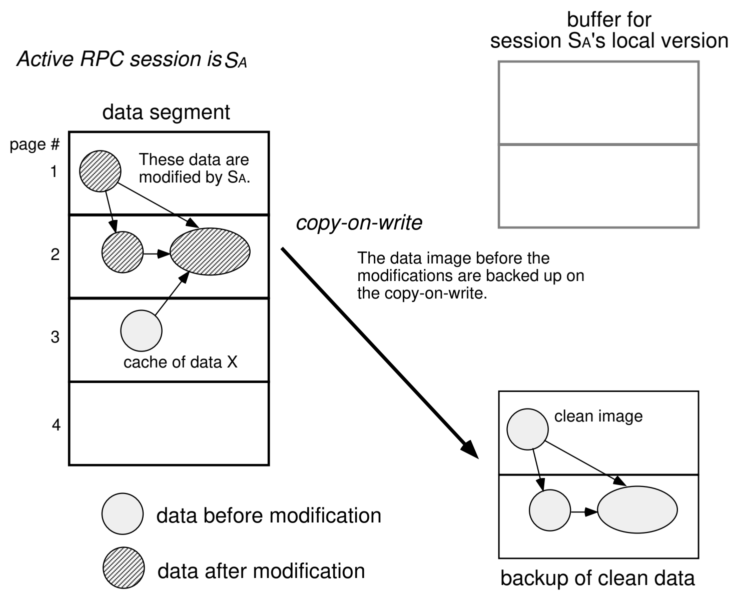


Figure 5: RPC session S_A updates pages #1 and #2 and creates a cache of the remote data X on page #3. A backup of pages #1 and #2 are made on the copy-on-write basis.

to the protocol used in our implementation as the *inter-session* protocol. Owing to the serializability of RPC sessions, concurrency is controlled properly.

In optimistic concurrency control techniques, no checking is done while an RPC session is executing [15]. In order to allow this, updates in an RPC session are *not* applied to original data until the RPC session reaches its end. During RPC session execution, all updates are done to a *local version* of the data that is kept for the RPC session. At the end of RPC session a validation phase checks whether any of the updates done in that session violate serializability. If serializability is not violated the local version is written to the original data and is made accessible to other RPC sessions; otherwise, the RPC session is aborted and restarted at a later time. Note that the inter-session protocol guarantees the *at-most-once* semantics of the RPC. If an RPC session fails to reach its end, the runtime system aborts the operations performed by that session.

To implement an optimistic concurrency control, we had each RPC session keep its own local version of data. To create a local version of data, it seems that it would be sufficient to create a copy of memory pages for each RPC session on the copy-on-write basis. When an attempt to modify a page is made, the runtime system would then create a copy of the page and modify the copy. Unfortunately, this approach does not work properly in our system. Since there may be some references into the accessed page from outside the page, the location (address) of the page cannot be moved. In other words, the location of the copy must be the same as that of original memory page.

Our solution to this problem is as follows. When there is an attempt to modify a page, it is detected by the MMU and the runtime system creates a *backup* of the clean (not modified) contents of the accessed page. After that, the memory protection of the accessed page is released and the accessed page itself is modified. While the modified page plays the role of a local version, the backup of the clean contents plays the role of an original version. When an active RPC session switches, the dirty (modified) page is copied into a buffer for a local version of the previously active session and the clean (not modified) contents of the page that have been backed up are loaded into the page's location. Thus the transient image of dirty pages is hidden from the newly activated RPC session, which observes clean image of the page.

To reduce the cost of creating local versions, our implementation divides the data segment under our system control into two sections: one section is a group of pages on which only cached data is allocated, and the other is a group of pages on which only original (not cached) data is allocated. Although a local version of original data is created in the fashion described above, a local version of cached data is created differently. In our implementation, the cached data itself is the local version in each RPC session because

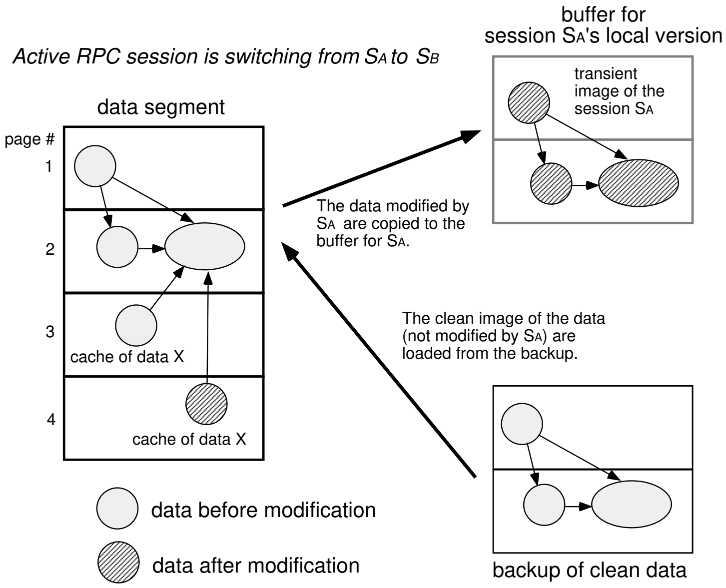


Figure 6: The active session switches from S_A to S_B. The modified contents of pages #1 and #2 are copied to S_A's local buffer, and the clean image of the pages is loaded from the backup. S_B creates its own cache of the data X on page #4 on a per-session basis.

each RPC session creates a cache of the data on a *per-session* basis. That is, each RPC session creates its own caches even if they are the caches of the same data; RPC sessions do *not* share any caches. Hence each RPC session has its own image of caches, and we can avoid creating a local version of cached data explicitly.

Figure 5 shows the situation where an RPC session S_A is executing. The pages #1 and #2 are for original data, and the pages #3 and #4 are for caches. The RPC session S_A creates a cache of the remote data X on page #3 and modifies pages #1 and #2. Before the modifications, the clean image of these pages is backed-up on a copy-on-write basis. Figure 6 shows the situation where the active RPC session switches from S_A to S_B. The local version of S_A is copied into S_A's buffer. Then the clean (not modified) contents of pages #1 and #2 are loaded from the backup. Thus the session S_B need not observe S_A's transient image of pages #1 and #2. Session S_B creates a cache of the remote data X on page #4. Although S_A has already cached X, S_B creates its own cache of the data X on a per-session basis. Session S_B updates the cache, but S_A's cache remains clean. Figure 7 shows the situation in which S_A is invoked again. In order for the computation executed by S_A to be resumed, S_A's local version is loaded to pages #1 and #2.

The violation of the spatial constraint imposed by the RPC session model is detected by the runtime system. In the situation shown in Fig. 7, if S_A attempts to transmit a reference to S_B's cache, the runtime system can detect it because the system fails to unswizzle the address of S_B's cache; S_A's data allocation table has no entry for S_B's cache. When a violation is detected, the runtime system raises a runtime exception.

At the end of an RPC session, the modified data is written back to its birthplace. Then the validation phase checks whether other RPC sessions have updated the memory pages the validating RPC session accessed. If and only if no updates were performed, the RPC session is validated and the local version of the session is written to the original locations. If the RPC session is not validated, it is rolled back by releasing its local version. The inter-session protocol schedules RPC sessions in the order in which an RPC session that finishes fastest is scheduled first. This protocol is simple, but does not pay much attention to increasing the degree of concurrency. This design choice is based on the observation that in cooperative autonomous systems most RPC sessions run independently of each other.

To prevent faulty RPC sessions from violating other sessions, we have an alternative to protect memory pages that should not be read or modified by the currently active RPC session. In Fig. 7, for example, page #4 should be protected to keep S_A from accidentally updating it. This implies a huge

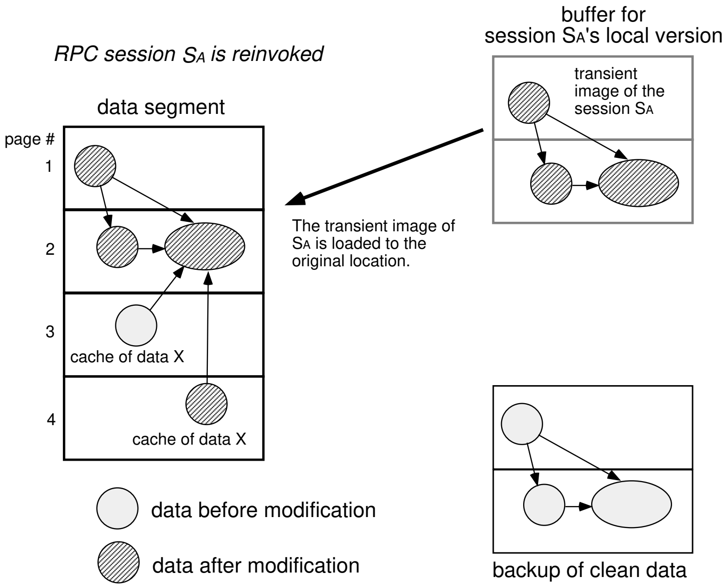


Figure 7: The RPC session S_A is activated again. The local version of S_A is loaded to pages #1 and #2.

overhead for changing page protections every time an active RPC session switches. If page protections are not changed, the performance becomes better but RPC sessions cannot survive a faulty RPC session. If applications are coded in type safe languages, these faulty accesses are statically checked by compilers. Our current implementation does not change page protections for better performance at the time of session switches.

5.2 Intra-Session Protocol for Data Coherency

To guarantee the data-sharing semantics of references, the TRAP system maintains the coherency between the original data and its caches. Since each RPC session possesses its own local version of shared data, it is sufficient to consider coherency inside each local version belonging to an RPC session; the coherency protocol guarantees that the only activity running within an RPC session observes a value assigned by the most recent assignment within that RPC session. Our coherency protocol, called the *intra-session* protocol, assures the data-sharing semantics within an RPC session.

There are many alternative protocols to assure data-sharing semantics. In principle, the intra-session protocol keeps track of the correspondence between a reference and the up-to-date value of the data designated by the reference. Which protocol is appropriate depends on the access pattern of applications. The working-set model [13] has been widely used to analyze the locality of accesses. In our execution model, since there is only one activity within an RPC session, the working-set model is effective. There are three phases in one RPC session: the *hot* phase, the *transition* phase, and the *cold* phase. When an RPC session starts, all the updated data will be in the working-set. In this phase, updated data will be accessed even after migration of an activity since it is in the current working-set. We call this phase the *hot* phase. As the computation proceeds, the working-set will be changing. In this phase some data updated in an RPC session will still be in the working-set, but other updated data might not be. Since the working-set is in transition, we call this phase the *transition* phase. After this transition phase the working-set completely changes. In this phase the current working-set contains data completely different from the data in the previous working-set. We call this phase the *cold* phase.

There is an appropriate protocol for each phase. In the hot phase, when an activity migrates to a remote address space, all the data updated within the RPC session should be transferred to the remote address space together with the other RPC arguments, since the updated data will be accessed in that address space. We refer to the modified data transferred together with RPC arguments as the *up-to-date value set*. This protocol that transfers the up-to-date value set is called the *eager coherency* protocol since it transfers up-to-date values eagerly. In this protocol the size of the up-to-date value set increases throughout an RPC session.

In the transition phase the data updated in the RPC session may or may not be accessed in the remote address space. It would be thus better to transfer an *update log* that records the hot location of the modified data (the address space from which the up-to-date value is available) since the update log is usually smaller than the up-to-date values themselves. We refer to this protocol as the *lazy coherency* protocol. In this protocol all the data registered in the update log is invalidated at the remote address space. When an attempt to access invalidated data is made, the up-to-date value of the data is fetched from its hot location. Thus up-to-date values are transferred lazily. In this protocol the size of the update-log increases throughout an RPC session.

In the cold phase the data updated in the RPC session is out of the current working set. It should be written back to its birthplace because it will not be accessed at remote address spaces. Since up-to-date values are scattered all over the participants of the RPC session, this writing back often incurs a huge overhead. But once it is written back there is no need to carry around an up-to-date value set or an update log, since up-to-date values are always available from birthplaces. We refer to this protocol as the *write-back coherency* protocol. After this protocol runs, all caches within the RPC session must be invalidated because they might be out of date. We cannot reuse cached data after this protocol runs.

From the above observation, the intra-session protocol combines the eager coherency, the lazy coherency, and the write-back coherency protocols. As the computation proceeds, our implementation switches from the eager coherency protocol, to the lazy coherency protocol, and finally to the write-back coherency protocol. Once the write-back coherency protocol is invoked, the sizes of both the up-to-date value set and the update log become zero. Then we restart the eager coherency protocol. The timing of protocol switching depends on the size of the working-set. An application programmer gives the runtime system a hint of the working-set size by specifying two protocol parameters: the *protocol eagerness* parameter and the *protocol laziness* parameter. The protocol eagerness parameter specifies the maximum size of an up-to-date value set, and the protocol laziness parameter determines the maximum size of an update log. A compiler should determine these parameters by analyzing a source program, but this would be difficult.

Our implementation uses the MMU hardware again to detect an update operation on shared data. When an update is detected by the MMU, the runtime system identifies the page on which the update is detected. Then the page protection is released so that the computation proceeds normally. The runtime system maintains a linked list of dirty (modified) pages in LIFO order. This linked list is used to determine the approximate order of updates; a page modified more recently is linked ahead. When an activity attempts to migrate to another address space, this linked list is examined from the head to the tail in order to create an up-to-date value set and an update log. An up-to-date value set is created by marshaling the data on the pages registered in the link into a network message, until the total size of the marshaled data reaches the protocol eagerness parameter. When the total size of marshaled data reaches the protocol eagerness parameter, an update log is created for the rest of the data on the pages registered in the list until the size of the update log reaches the protocol laziness parameter. When the size of the update log reaches the protocol laziness parameter, the write-back coherency protocol is invoked to write all the modified data to its birthplace. The system multicasts a message to all the session participants in order to direct them to write back the up-to-date values.

On the receipt of a network message, the runtime system decodes the data in the up-to-date value set in the message. If remote references are embedded in the values, they are decoded (swizzled) as described in Section 4.1. The memory pages on which the data registered in the update log resides are made unaccessible by protecting those pages. The hot location column of the DAT described in Section 4.2 is modified to the hot locations registered in the update log. When an attempt to access one of those unaccessible pages is made, a page fault occurs and the up-to-date values are transferred from the hot locations in the way described in Section 4. When an RPC session reaches its end, the write-back coherency protocol always runs to write back all the up-to-date values to their birthplaces. Then the runtime system disposes of all the caches created during the RPC session, since remote references (references to caches) are not valid outside RPC sessions.

6 Language Preprocessing

To provide higher transparency for application programmers, Trap cooperates with language preprocessors as conventional RPC systems do. A language preprocessor generates *stubs* that conceal communication details, although the runtime mechanism itself is language-neutral. The only assumption the Trap runtime system makes is that the front-end language is statically typed so that stubs can be emitted statically. The currently provided preprocessor translates Trap-C, our extension of ANSI/C, into the corresponding C code with requisite stubs. Other statically typed languages such as C++ and Java can be used as a front-end language.

To explain the interface offered to application programmers, Section 6.1 presents some sample codes in Trap-C. One notable feature of Trap-C is that it enables even "function" pointers to be passed as RPC arguments. Remote function pointers are primarily handled by a language preprocessor, while references to ordinary data are treated by the runtime system. Section 6.2 explains the algorithm used to generate stubs capable of passing remote function pointers. A second notable feature of Trap-C is that it allows application programmers to customize an algorithm to pickle a data structure for network transfer. We refer to a module that implements this algorithm as a *pickling module*. A pickling module is quite significant in the adaptive caching scheme, since it controls the eagerness of the adaptive caching scheme; that is, it determines the "shape" of a closure that is eagerly transferred to a remote address space. Section 6.3 presents an example of a customized pickling module.

6.1 Overview of Trap-C

The extensions of Trap-C are six keywords: `import`, `export`, `trap_session`, `trap_new`, `trap_delete`, and `remote`. The keyword `import` declares a function to be imported and the keyword `export` to be exported. The primitive `trap_session()` declares an RPC session (the code enclosed in `trap_session()`). In the following, the function `insert_node()` is imported and the function `traverse_node()` is exported.

```
typedef struct _tree_t {
    int key;
    struct _tree_t *left, *right;
} node_t;
import tree_t* insert_node(tree_t*);   /* import a procedure */
export tree_t* traverse_node(tree_t*); /* export a procedure */
...
trap_session(){                        /* declare an RPC session */
    tree_t* node = trap_new tree_t;    /* allocate a data */
    insert_node(node);                 /* remote call */
    trap_delete(node);                 /* reclaim the data */
}
```

The binding between imported and exported functions is outside the scope of the present paper. In our implementation, the locations of exported functions are available from a specific file or from a Trap name server. The bindings are resolved in the three stages: compile-time, startup-time, and run-time. If an imported function is not resolved at the compile-time, the binding is deferred until the startup-time. If the function is still not resolved at the startup-time, the binding is further deferred until there is an attempt to invoke the imported function. If the binding cannot be resolved at that time, a runtime exception is invoked.

The primitive `trap_new` allocates data within a data segment managed by Trap. For data in the Trap's data segment to be reclaimed, the primitive `trap_delete` is provided. If we want to prevent some specific data from being remotely referenced, we simply allocate the data in a usual data segment rather than in the Trap's data segment. The keyword `remote` is used to declare a remote "function" pointer. The details of remote function pointers are described in Section 6.2. The Trap-C preprocessor translates these keywords into normal C code and generates requisite stubs. After the translation, the back-end compiler (a usual C compiler such as `gcc`) compiles the translated code and stubs, and links them with the Trap runtime library to emit executables.

6.2 Stub Generation and Remote Function Pointers

The Trap-C translator generates stubs for imported or exported procedures. In the following, two procedures `tree_lookup()` and `tree_traverse()` are imported.

```
/* import procedures */
import int tree_lookup(tree_t* tree, int key);
import int tree_traverse(tree_t* tree, int num_of_hops);
```

The TRAP-C translator generates caller stubs as shown below. At line 4, `get_locator()` looks up a table that records the binding of imported procedures and their physical locations. If a corresponding entry is not found in the table, the runtime system inquires of the TRAP name server which address space exports the target procedure. The value `locator` returned by `get_locator()` contains sufficient information to locate a remote procedure: (1) an identifier of the address space exporting the remote procedure, (2) an address of the callee stub, and (3) an address of the procedure body. In this example, both stubs invoke the procedure `tree_int_to_int()` which is generated by the TRAP-C translator. The procedure `tree_int_to_int()` marshals the arguments, invokes the remote procedure specified by the locator, and unmarshals the results. The procedure `tree_int_to_int()` is a *generic* caller stub in the sense that it can operate on all procedures of the type $tree\_t* \times int \rightarrow int$.

```
1: /* a caller stub of tree_lookup() */
2: int tree_lookup(tree_t* tree, int key){
3:     /* get the location */
4:     function_locator_t locator = get_locator("tree_lookup");
5:     return tree_int_to_int(locator, tree, key);
6: }
7: /* a caller stub of tree_traverse() */
8: int tree_traverse(tree_t* tree, int key){
9:     function_locator_t locator = get_locator("tree_traverse");
10:    return tree_int_to_int(locator, tree, key);
11: }
```

The code of `tree_int_to_int()` is shown below. At line 5, the procedure `marshal_intra()` marshals additional data for the intra-session protocol, such as an up-to-date value set and an update log. In the next line, a closure of the passed tree is created by a pickling module indicated by `pickle_tree()`. Then an integer argument is marshaled, an RPC is invoked, and the result is unmarshaled. The procedure `unmarshal_intra()` unmarshals the additional data for the intra-session protocol.

```
1: /* generic caller stub of tree_t* × int → int */
2: int tree_int_to_int(function_locator_t locator, tree_t* tree, int key){
3:     message_buffer_t args, results;
4:     /* marshal data for the intra-session protocol */
5:     marshal_intra(&args);
6:     (*pickle_tree)(tree, &args);          /* marshal a tree */
7:     int_encode(key, &args);               /* marshal an integer */
8:     remote_call(locator, &args, &results); /* invoke an RPC */
9:     int_decode(&r, &results);             /* unmarshal an integer */
10:    /* unmarshal data for the intra-session protocol */
11:    unmarshal_intra(&results);
12:    return r;
13: }
```

To export the procedures `tree_lookup()` and `tree_traverse()`, the server program declares them to be exported as follows. The TRAP-C translator generates for these procedures a generic *callee* stub that unmarshals the arguments, invokes the target procedure, and marshals the results.

```
/* exporting procedures */
export int tree_lookup(tree_t* tree, int key);
export int tree_traverse(tree_t* tree, int num_of_hops);
```

```
/* generic callee stub */
void tree_int_to_int(function_locator_t locator, message_buffer_t* args){
    message_buffer_t results;
    tree_decode(&tree, &args);          /* unmarshal a tree */
    int_decode(&key, &args);            /* unmarshal an integer */
    unmarshal_intra(&args);
    /* invoke the function body */
    r = (*(int(*)(tree_t*,int))locator.function_body)(tree, key);
```

```
    marshal_intra(&results);
    int_encode(r, &results);           /* marshal the return value */
    remote_return(&results);           /* return from RPC */
}
```

To use a remote "function" pointer, an application programmer must qualify a type declaration of a function pointer with `remote` qualifier. The translator handles remote function pointers. In the following code, a function `f` is declared to be a remote pointer to a function of the type $tree\_t* \times int \rightarrow int$. The function `f` is initialized to indicate a remote function by receiving a function pointer as a result of a remote procedure call. To invoke a remote function through a remote function pointer, we can use the same syntax that is used with the local calls.

```
int (*remote f)(tree_t*, int); /* declare a remote function pointer */
/* call a remote procedure that returns a function pointer */
f = get_remote_function();
x = (*f)(root, 123);           /* invoke the remote function */
```

The TRAP-C translates the above code into the following code. The translator generates a generic caller stub for the procedures of the type $tree\_t* \times int \rightarrow int$. Then it translates the declaration of `f` into a code that declares a pointer to a function locator. Specifically, a remote function pointer is represented by a pointer to a function locator that locates a referenced function. To make it possible to transmit a function pointer, the pointer is encoded into a corresponding function locator. Then the translator converts an invocation of a remote function through a pointer into an explicit invocation of a corresponding generic caller stub. Since `f` indicates a function locator that locates the function referenced by `f`, the generic caller stub uses `*f` as a function locator. This translation algorithm is based on an algorithm presented by Ohori and Kato [32]. Their translation algorithm is powerful in the sense that it can handle polymorphic higher-order functions. We use the monomorphic portion of their algorithm.

```
/* generic caller stub */
int tree_int_to_int(function_locator_t locator, tree_t* tree, int key){
    /* same as the previous example */
}
/* translated code */
function_locator_t* f;           /* f denotes a function locator */
f = get_remote_function();       /* not changed */
/* explicit call to the generic stub */
x = tree_int_to_int(*f, tree, 123);
```

The generic callee and the caller stubs of the procedure `get_remote_function()` operate as follows. In the callee stub, a function pointer is converted into a corresponding function locator and then marshaled into a network message. In the caller stub, the received function locator is decoded and a pointer to the decoded function locator is returned.

```
/* generic callee stub */
typedef int (*func_t)(tree_t*, int);
void void_to_func(function_locator_t locator){
    func_t r;
    r = (*(func_t(*)())locator.function_body)();
    /* r is converted into a function locator in function_encode() */
    function_encode(&r, &results);
    remote_return(&results);
}
```

```
/* generic caller stub */
function_locator_t* void_to_func(function_locator_t locator){
    function_locator_t* r;
    remote_call(locator, NULL, &results);
    /* r is assigned a pointer to a function locator that specifies the target */
    function_decode(&r, &results);
    return r;
}
```

6.3 Customizable Pickling Module

As described in Section 4.3, a pickling module plays a significant role in the adaptive caching scheme of Trap, since it determines to what extent the transitive closure of a reference is transferred eagerly. The default pickling module creates a certain depth of a transitive closure in the breadth-first manner. Application programmers can change the depth by specifying a *closure size* parameter that bounds the amount of a transferred closure. In addition, they can replace the module with their own to customize an algorithm to make a closure. An application programmer who knows the access pattern of a server process can customize the module according to the access pattern. For example, if the server is known to traverse the transmitted closure in the depth-first manner rather than in the breadth-first manner, the pickling module can be customized to create a closure in the depth-first manner.

There are two operations on a reference to create a closure transitive from a given reference. The first operation is to marshal data referred to by the given reference. The second operation is to obtain a set of references directly transitive from the given reference. To specify a traverse algorithm, we determine the order in which a reference is traversed and which referenced data is marshaled. In the pre-order traverse, we first marshal the referenced data and then traverse the reference. In the post-order traverse, we first traverse the reference and then marshal the referenced data. In our implementation, a pickling module determines the order of marshaling and traversing references. In the following, we show an example that creates a closure of a binary tree in the depth-first manner. For ease of understanding, this example is simplified. As described in Section 7.1, we have implemented several pickling modules to confirm the interface of our design.

```
void depth_first(tree_t* tree, message_buffer_t* buf){
    if(reference_check(tree)){          /* check validity of reference */
        tree_marshal(tree, buf);        /* marshal a node */
        /* recursively marshal child nodes */
        depth_first(node->left, buf);
        depth_first(node->right, buf);
    }
}
```

In `depth_first()`, the validity of a reference is checked at first by calling `reference_check()`. The `reference_check()` returns false if a reference is null or refers to invalid data, such as an empty cache or invalidated data. The `reference_check()` also checks whether the referenced data is already marshaled into a message. If it is, it returns false to keep the system from falling into an infinite loop when marshaling a cyclic structure. It also returns false if the size of the data already marshaled exceeds the closure size parameter. In `depth_first()`, if the reference is valid, the referenced node is marshaled and the child nodes are recursively marshaled. Application programmers with little knowledge of the internals of the Trap runtime system can customize a pickling module by using the library provided by the Trap runtime system.

7 Experiments

To study the performance of the VM-integrated RPC, we implemented our prototype Trap system on Sun Sparc workstations and PCs. The workstations were equipped with UltraSPARC (200MHz) processors and 160 Mbytes of main memories, and they were running Solaris 2.5.1. The PCs were equipped with Intel PentiumII (300MHz) processors and 128 Mbytes of main memories, and they too were running Solaris 2.5.1. Our implementation does not use any system calls specific to Solaris, so it will be able to be ported to other operating systems. UltraSPARC and PentiumII processors use different data formats. For example, UltraSPARC processors use a big-endian representation, whereas PentiumII processors use a little-endian representation. In UltraSPARC, word accesses must be aligned on a 4-byte boundary, whereas PentiumII does not. The size of virtual memory page of UltraSPARC is 8 Kbyte, whereas that of PentiumII is 4 Kbyte. These machines are connected together by 10Mbps Ethernet. TCP/IP is used as a transport layer protocol. Our implementation converts data representations according to target architectures. As a canonical data representation, we used the XDR (eXternal Data Representation). The Trap-C translator uses `gcc` (version 2.7.2) as the back-end C compiler.

We performed some experiments using the TRAP system. It employs the adaptive caching scheme and allows the customization of pickling modules. Section 7.1 compares our adaptive caching scheme with the other schemes described in Section 2. Section 7.2 shows that the overhead of presence tests is not negligible in computation-bound benchmarks. Section 7.3 examines the performance of the intra-session protocol, and Section 7.4 estimates the overhead of the inter-session protocol.

7.1 Comparison of Alternative Caching Schemes

To model data structures for representing shared information, our experiments used a collection of 16 clusters of objects, each cluster having exactly two connections to other clusters (we arranged for all clusters to be connected). Each cluster was indexed by an integer from 0 through 15 and comprised 64 inter-connected objects, each composed of 28 integers and 4 references to objects within the same cluster. Each cluster is 8 Kbytes in size, so the total size of the collection of the clusters was 128 Kbytes.

7.1.1 Comparison of caching schemes

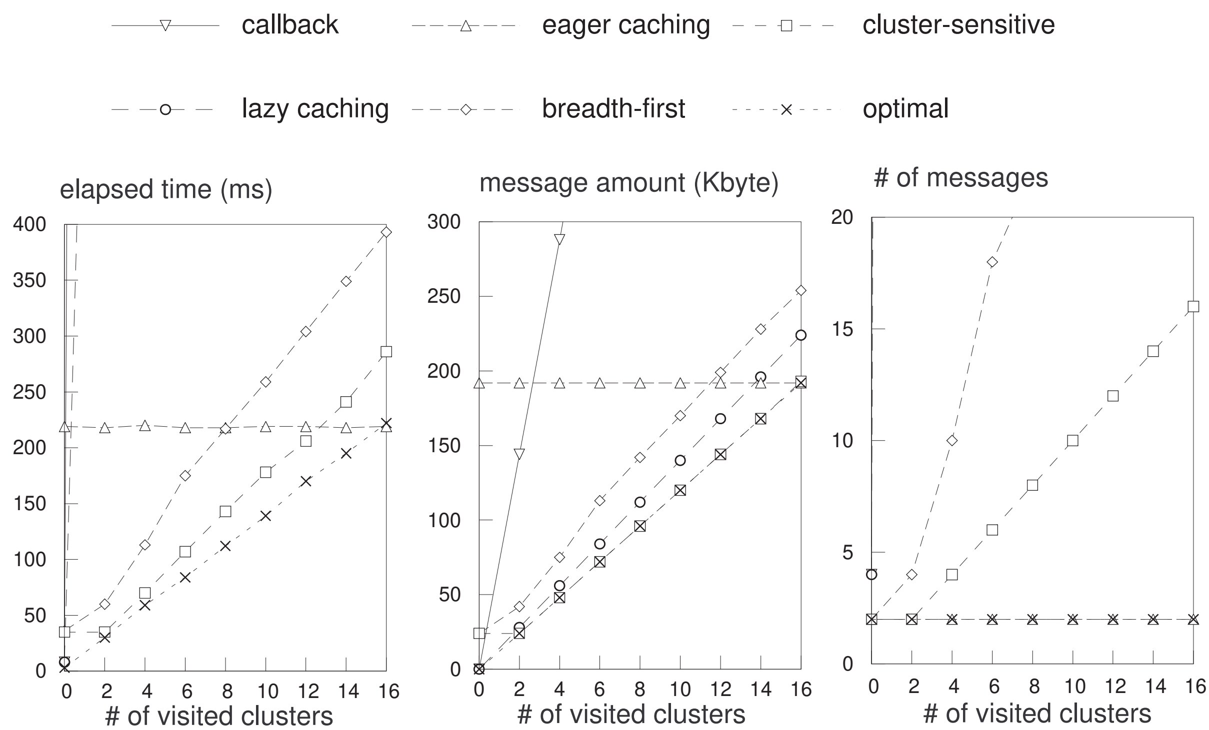
To compare the VM-integrated RPC with the other caching schemes, we measured the performance of a communication-bound benchmark in which the data structure described above was *remotely* traversed. In this experimental subject, a client generates a collection of the clusters and passes to a server a reference to the cluster of index 0. Then the server *remotely* visits all the objects within the cluster of index 0, then visits the objects within the cluster of index 1, and so on. We varied the number of visited clusters and measured the elapsed time at the client for visiting the clusters.

scheme	description
callback	The callback scheme. Every dereference of a remote reference requires a callback.
lazy caching	The lazy caching scheme. Remote data are cached lazily or on demand.
eager caching	The eager caching scheme. The entire collection (128 Kbytes in size) of clusters is always transferred and cached at once by the server.
breadth-first caching	Our adaptive caching scheme. A transferred closure is taken in the breadth-first manner. The closure size parameter is set to 16 Kbytes. This pickling-module is *not* conscious of clustering; it follows a reference to a cluster other than the cluster of interest.
cluster-sensitive caching	Our adaptive caching scheme. The pickling module is customized to be aware of clustering; it is tailored to marshal the objects only within the cluster of interest. The closure size parameter is set to 16 Kbytes.
optimal caching	Our scheme. This scheme is optimal in the sense that it is tailored to transfer at once all the objects visited by the server.

Table 4: Caching schemes compared in the experiment.

In this experiment, we compared the callback scheme, the lazy caching scheme, the eager caching scheme, and our adaptive caching scheme. In the adaptive caching scheme we customized a pickling module, and in this experiment we used three pickling modules: (1) the default pickling module, (2) a cluster-sensitive pickling module, and (3) an optimal pickling module. The first default pickling module created a transferred closure in the breadth-first manner. The closure size parameter, which bounds the maximum size of the transferred closure, was set to 16 Kbytes. Although the server process visited each cluster in sequence, the default pickling module was not aware of clustering and transferred objects outside the cluster of interest. The cluster-sensitive pickling module, on the other hand, was conscious of clustering and marshaled every object within each cluster. In this experiment, since the closure size parameter was set to 16 Kbytes, two clusters were transferred at once (each cluster is 8 Kbytes in size). The optimal pickling module was optimal in the sense that it was tailored to transfer at once all the objects that the server actually visited. The performance of this optimal pickling module shows the best possible performance of our current implementation.

Table 4 summarizes the caching schemes used in the experiment, and Fig. 8 shows the experimental results. Figure 8 (a) shows the elapsed time. The callback and the lazy caching schemes are apparently uncompetitive with the other schemes. Since each object is visited just once in this experiment, the caching effect of the lazy caching scheme is poor. Figure 8 (b) shows the amount of network data transfer, and Figure 8 (c) shows the number of network communications. In the case of the lazy caching scheme, the amount of network data transfer was not large, but the increased number of network communication degraded the performance. Although both the lazy caching and the optimal schemes transfer only the objects visited by the server, the lazy caching scheme transfers more network messages than the optimal scheme does. This is because the lazy caching scheme requires additional messages to request the client to transfer the accessed object.



(a) elapsed time (msec) (b) amount of messages (Kbyte) (c) number of messages

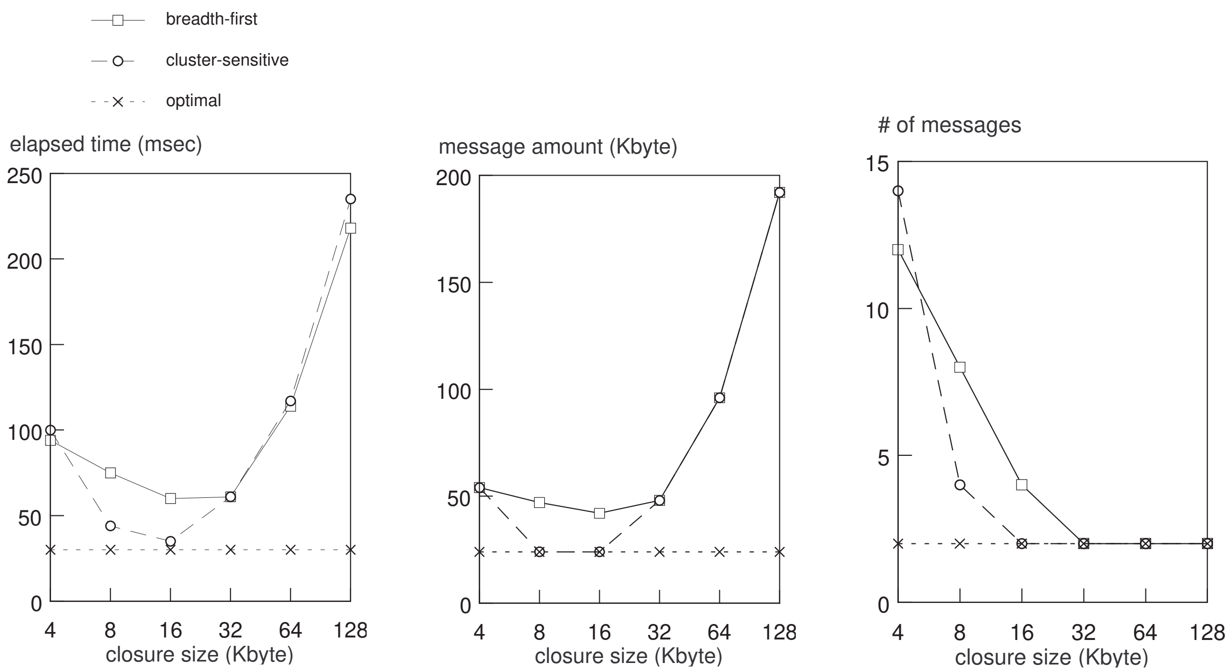
Figure 8: Comparison of alternative caching schemes. The server program *remotely* traverses a collection of clusters created at the client side. X-axes show the number of visited clusters. Y-axes show the elapsed time, the amount of data transfer, and the number of network communications.

The eager caching scheme shows almost constant time since it replicates all the clusters at once. When the number of visited clusters is small, this scheme shows poor performance, but its performance becomes better as the number of visited clusters increases. Figure 8 (a) shows that when more than eight clusters are visited the eager caching scheme performs better than our breadth-first and cluster-sensitive schemes do. This does not, however, mean that our schemes are not appropriate for these cases: we can obtain the performance of the eager caching scheme simply by setting the closure-size parameter to infinity. Comparing the breadth-first and the cluster-sensitive schemes, we see that the cluster-sensitive scheme reduces both the amount of data transfer and the number of network communications. This is due to the customization of a pickling module. This customization improves performance by 20 to 50 percent. The optimal scheme shows the best performance since the pickling module “knows” the access pattern of the server.

7.1.2 Effect of customizing pickling modules

To evaluate the effect of customizing pickling modules, we examined the performance of the breadth-first, the cluster-sensitive, and the optimal pickling modules. In this experiment we changed closure size parameters and measured the performances of the three pickling modules. In this experiment, two clusters were remotely visited in the same way as the previous experiment. The closure size parameters in the breadth-first and cluster-sensitive schemes were set to 4, 8, 16, 32, 64, and 128 Kbytes. In the optimal pickling module, the closure size parameter is meaningless since this module always transfers the precise subset of the transitive closure.

Figure 9 shows the results. X-axes show the closure-size parameter in the logarithmic scale and Y-axes show the elapsed time in milliseconds, the amount of data transfer, and the number of network messages. The line of the optimal module in the figure indicates the best performance (although the closure size parameter is meaningless in the optimal pickling module). In Figure 9 (a), both the breadth-first and the cluster-sensitive pickling modules show the best performance when the closure size parameter is 16 Kbytes. This is reasonable, since the total size of the visited clusters in this experiment is 16 Kbyte (two 8-Kbyte clusters). Comparing the breadth-first and the cluster-sensitive pickling modules, we see that the cluster-sensitive module shows better performance in many cases. From Figs. 9 (b) and 9 (c), we can see that the closure-sensitive module reduces both the amount of data transfer and the number of network communications. This is due to the fact that the cluster-sensitive module was tailored to the access pattern of the server program.



(a) elapsed time (msec) (b) amount of messages (Kbyte) (c) number of messages

Figure 9: Effect of customizing pickling modules. Two clusters are remotely visited using the breadth-first, the cluster-sensitive, and the optimal pickling modules. X-axes show the closure-size parameter on logarithmic scales. Y-axes show the elapsed time, the amount of data transfer, and the number of network communications.

7.1.3 Pickling overhead

When transmitting a data structure, our implementation of the VM-integrated RPC pickles the data structure in order to maintain the autonomy of cooperating processes. This pickling cost is frequently pointed out as a crucial drawback of RPC. In the following experiment, we examined the ratio of the

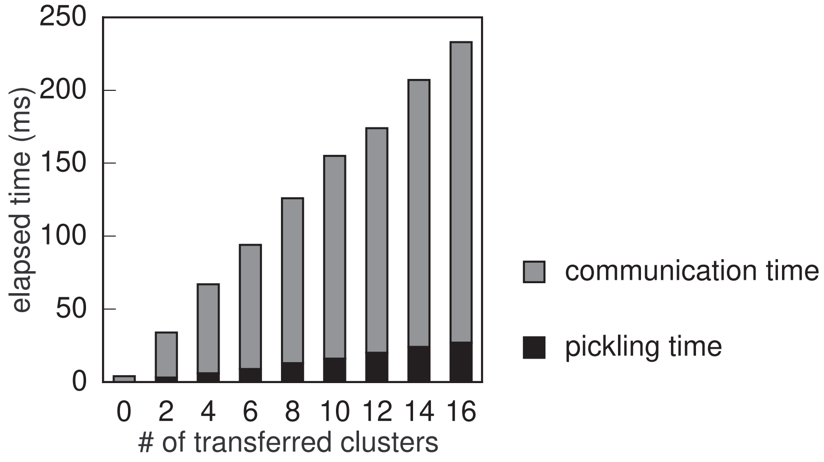


Figure 10: Cost of pickling and communication.

pickling cost to that of network communication. We used the same data structure in the previous experiments, varied the number of transferred clusters, and measured the pickling time and the communication time. We can see from Fig. 10 that the time needed for pickling is about 10% of the total communication time regardless of the number of transferred clusters. Since the number of network communications can be reduced if pickling modules are properly customized, the pickling cost would not much degrade application performance. If the bandwidth of networks is larger than that of the network we used (10Mbps Ethernet), the ratio of the pickling cost will rise and may become the performance bottleneck, but if communicating processes have the same architecture, we can reduce the pickling cost by omitting the conversion of data representations.

7.2 Comparison of Presence Test Schemes

As described in Section 2.1, the performance of computation-bound applications is limited by the cost of presence tests. Conventional implementations of remote references require presence tests to distinguish remote references from local references. In the VM-integrated RPC, since the MMU hardware detects an attempt to derefer a remote reference without software intervention, the cost of presence test is nothing (apart from the overhead due to a page fault). In the following experiment, we compared the VM-integrated RPC with the software-based presence tests: the tagged reference approach and the proxy-overlay approach. As mentioned in Section 2.2, the tagged reference approach tags remote references to distinguish them from local references. In the proxy-overlay approach, a proxy detects the first invocation on the proxy and overlays itself with the target remote object. The proxy-overlay approach needs a method to be invoked indirectly through a method table so as to replace the code of methods.

	VM-integrated RPC	tagged reference	proxy overlay
elapsed time (sec)	1.82	2.46	2.23
ratio	1.00	1.35	1.23

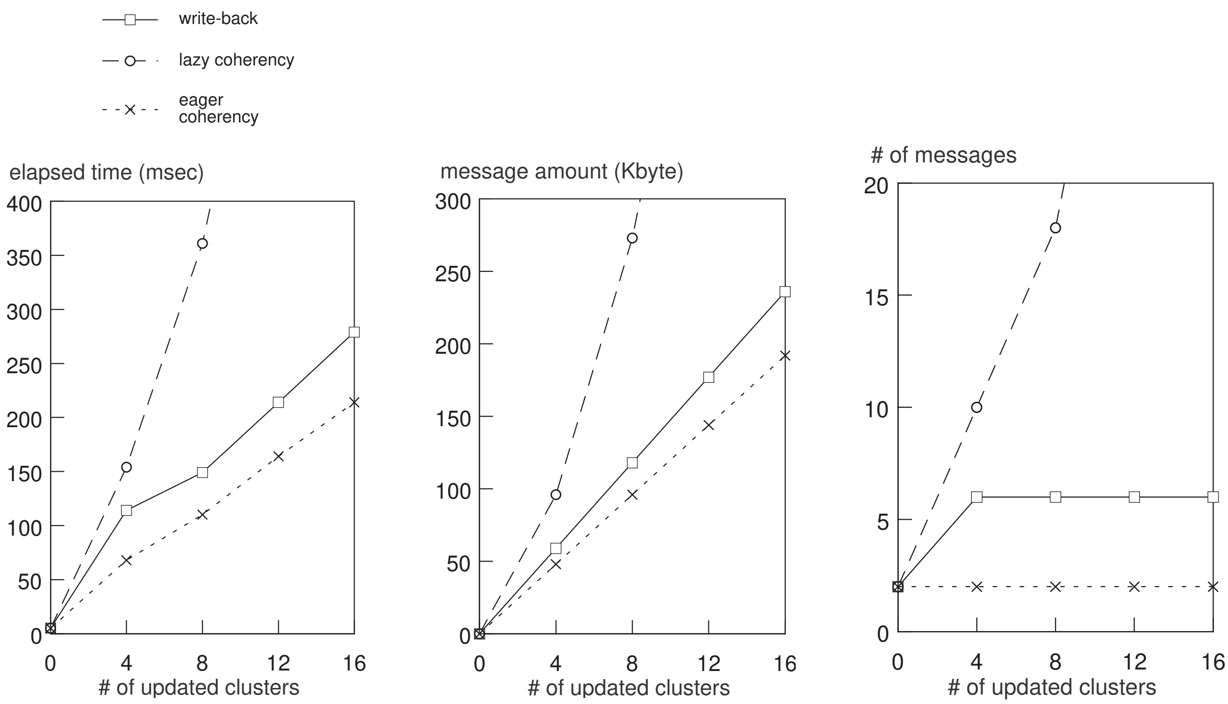
Table 5: Comparison of the overheads of presence tests.

In this experiment we used the same data structure used in the previous experiment and measured the time needed to traverse all the clusters with all the objects visited within each cluster. To emulate computation-bound applications, we cached all the clusters before the measurement. In measuring the overhead of presence tests, we repeated the traverse 10,000 times. In the tagged reference approach, the code for tag checking was inserted by hand. In the proxy-overlay approach, a method that traverses an object was invoked indirectly through a method table.

The times measured on a PC (PentiumII, 300MHz) are listed in Table 5. The results indicate that the software overhead of tag checking is not negligible; it increases the execution time by 35%. The proxy-overlay approach is worse than the VM-integrated RPC because of the additional overhead of invoking a method indirectly. Note that the proxy-overlay approach prohibits the compiler optimizations such as in-line expansion of methods. Thus it is not easy to improve the performance of applications that use the proxy-overlay approach. The performance of applications using the tagged reference approach, on the other hand, can be improved if a compiler is powerful enough to eliminate redundant tag checks. Hosking and Moss [20] compared the memory management techniques of object-oriented languages and reported that the tagged reference approach is competitive with the approaches that exploit MMU hardware. They used a Smalltalk compiler when they measured performance. In clean languages such as Smalltalk, a compiler can eliminate most redundant tag checks, but we believe that most languages (such as C++) will benefit less from compilers because of the difficulties of flow analysis.

7.3 Performance of Intra-Session Protocol

In the experiments described here so far, no update operations were performed. In the following experiments, however, we measure the performance of update operations. As explained in Section 5.2, the intra-session protocol is a combination of three coherency protocols: the eager coherency protocol, the lazy coherency protocol, and the write-back coherency protocol. First we examined the performance of each protocol in two benchmarks. One benchmark, called the *hot-update* benchmark, corresponds to the hot phase of the working-set model in which the data updated in one address space are also accessed in another address space. The other benchmark, called the *cold-update* benchmark, corresponds to the cold phase in which the updated data are not accessed in another address space. We also examined the effect of combining these protocols. To do this we used a benchmark, called the *transient-update* benchmark, in which a working-set changes. By specifying the protocol eagerness and the laziness parameters properly, our intra-session protocol adapts to the transition of working-sets.



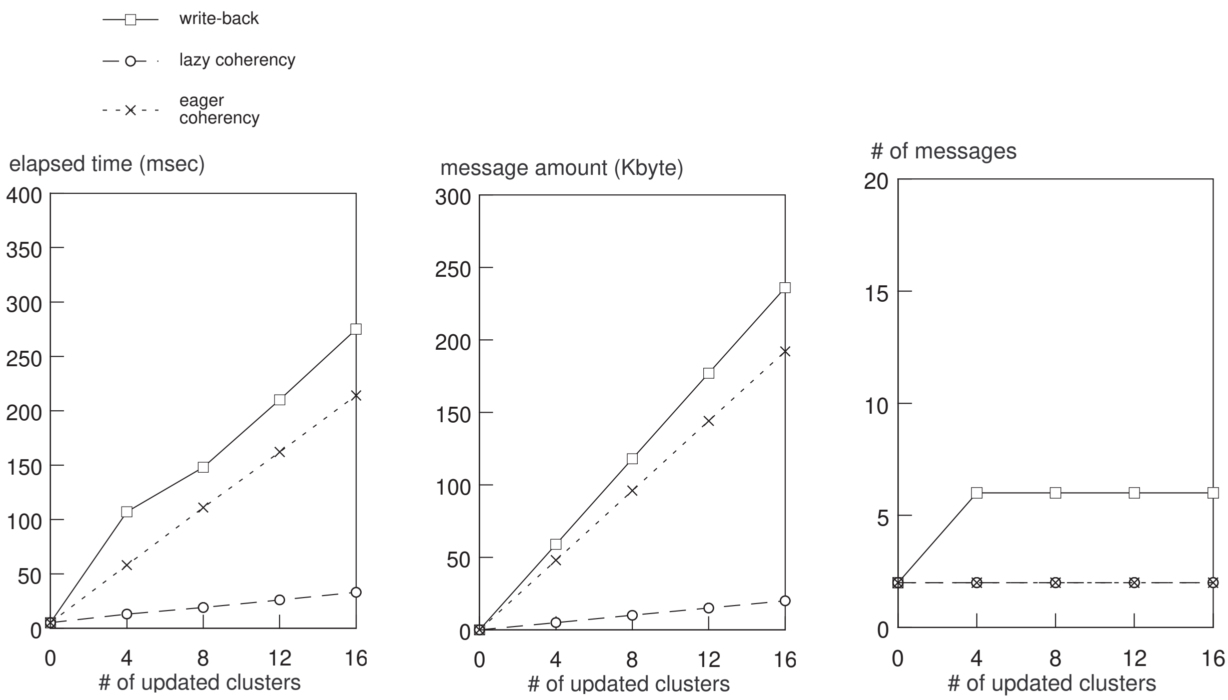
(a) elapsed time (msec) (b) amount of messages (Kbyte) (c) number of messages

Figure 11: Comparison of three protocols in the hot-update benchmark.

7.3.1 Comparison of three protocols

To examine the properties of the three protocols, we used the same data structures used in the previous experiments. In the hot-update benchmark, an activity migrates to a server and updates some clusters. Then it returns to the client and traverses all the clusters at the client address space. This corresponds to the hot phase. In the cold-update benchmark, on the other hand, after the activity returns to the client address space, it does not access any clusters. This corresponds to the cold phase. We varied the number of updated clusters and measured the performance of each benchmark. To discern the performance of the protocols, the entire clusters were duplicated ahead of time at the server address space.

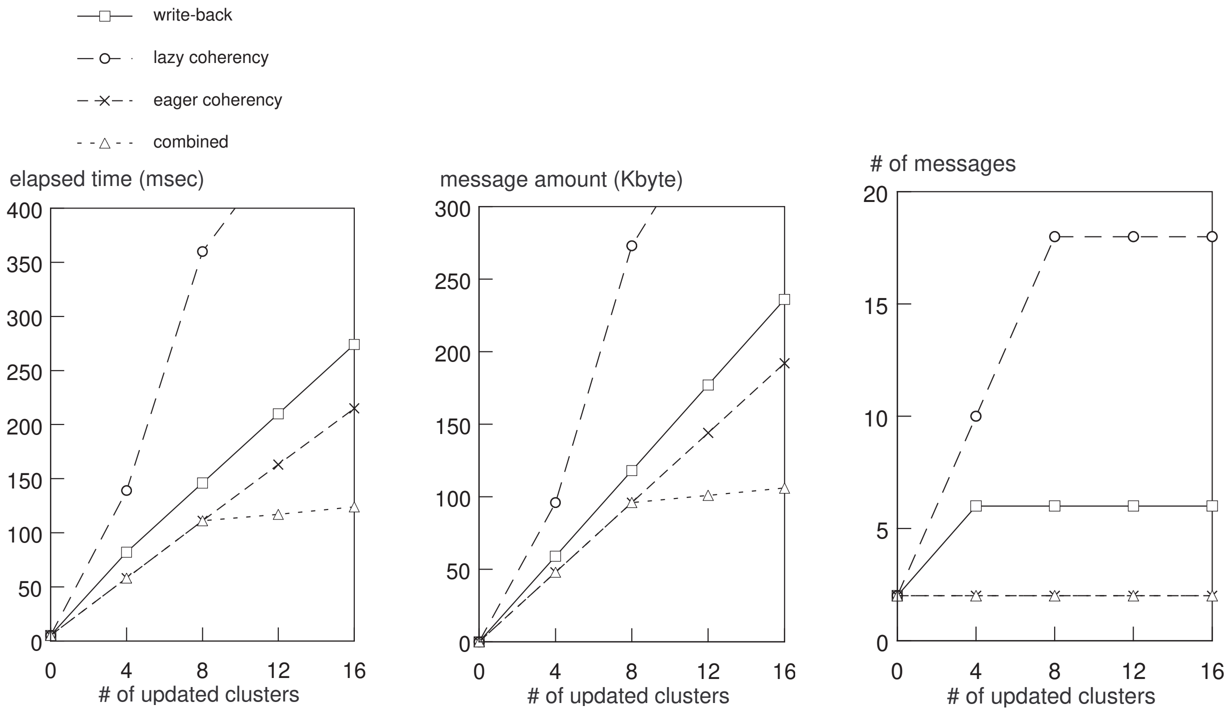
Figure 11 shows the results of the hot-update benchmark. X-axes show the number of updated clusters, and Y-axes show the elapsed time, the amount of data transfer, and the number of network communications. In Fig. 11 (a), the lazy coherency protocol shows poor performance. This is because it transfers an update log and then transfers accessed objects. Actually, both the amount of data transfer and the number of network communications are larger with the lazy coherency protocol than they are with the other protocols. In the hot-update benchmark, the eager coherency protocol shows better performance than the others, since accessed objects are transferred eagerly. The write-back coherency protocol shows performance competitive with that of the eager coherency protocol, but this protocol will show terrible performance if the activity tries to access the clusters cached at the server because this would invalidate all the caches.



(a) elapsed time (msec) (b) amount of messages (Kbyte) (c) number of messages

Figure 12: Comparison of three protocols in the cold-update benchmark.

Figure 12 shows the results of the cold-update benchmark. The eager coherency protocol and the write-back coherency protocol show the same performance that they did in the hot-update benchmark, since their runtime behaviors do not change according to the access pattern of the client. In this benchmark, the lazy coherency protocol shows the best performance of the three since the objects updated at the server are not accessed at the client. Compared with the eager coherency protocol, the lazy coherency protocol transfers a smaller amount of data because the size of an update log is usually less than that of an up-to-date value set. We can confirm this from Figs. 12 (b) and 12 (c).



(a) elapsed time (msec) (b) amount of messages (Kbyte) (c) number of messages

Figure 13: Effect of protocol combination.

7.3.2 Effect of combining the protocols

In a situation where the working-set changes during an RPC session, we should combine the three protocols. In the following experiment, we used the transient-update benchmark. This benchmark is the same as the hot-update benchmark except that only half of the clusters are traversed at the client. Therefore, if the number of updated clusters is eight or less (the total number of clusters is 16) we should use the eager coherency protocol. If the number of updated clusters exceeds eight we should use the lazy coherency protocol. By specifying the protocol eagerness and the laziness parameters, we can set up this protocol behavior. In this experiment we set the protocol eagerness parameter to 64 Kbytes and the protocol laziness parameter to infinity. We compared the performance of the three protocols with the protocol that combines the eager and the lazy coherency protocols.

As shown in Fig. 13, the three protocols show performance similar to that shown on the hot-update benchmark. When the number of the updated clusters is less than eight, the performance of the combined protocol is the same as that of the eager coherency protocol. When the number of the updated clusters exceeds eight, the combined protocol shows performance better than that of the eager coherency protocol. This is the effect of changing the protocol adaptively from the eager coherency protocol to the lazy coherency protocol.

7.4 Performance of Inter-Session Protocol

Since our implementation of the inter-session protocol is based on the optimistic approach, the runtime overhead of the protocol is expected to be relatively small. To confirm this we measured the total time needed to execute an RPC session in which an activity updates some clusters at the client, migrates to the server, and updates them again. We varied the number of the updated clusters. All the clusters were duplicated ahead of time at the server, and the protocol eagerness parameter was set to infinity. We broke down the measured time into four parts: (1) the read phase, (2) the backup phase, (3) the validation phase, and (4) the write phase. The read phase includes the time spent in the *intra*-session

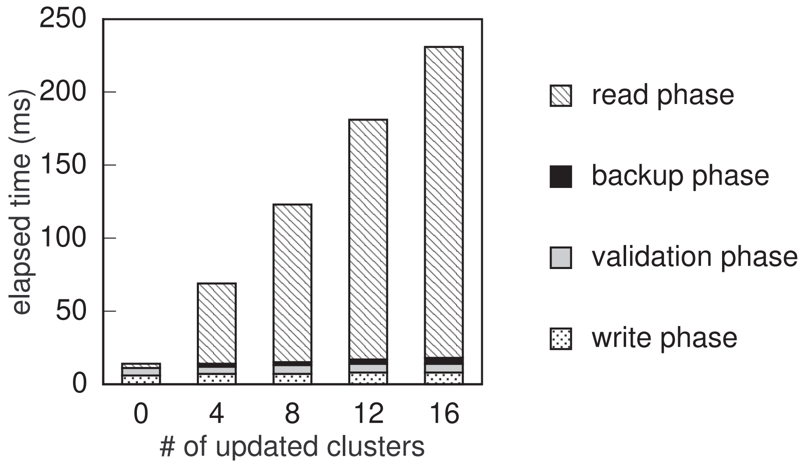


Figure 14: Cost of backup, validation, and write-back phases.

protocol. The phases other than the read phase are the time spent in the *inter*-session protocol. The backup phase is the time spent creating a local version of an RPC session. The validation phase, which is invoked at the end of an RPC session, is the time spent on checking the serializability of the RPC sessions. In this experiment the validation phase always succeeded because there was only one RPC session. The write phase is the time spent writing the local version of the RPC session back to the original locations.

We can see in Fig. 14 that when no clusters are updated, the total time required by the inter-session protocol is almost 80% of the elapsed time. This cost can be reduced if we introduce *read-only* RPC sessions, in which no update operations are allowed. As the number of updated clusters increases, the ratio of the cost of the inter-session protocol falls from 20% to 8%. When some clusters are updated, the cost of maintaining the data-sharing semantics dominates the entire performance. As a result, the relative overhead of the inter-session protocol is reduced. Thus, the most effective way to improve the performance of our system will be to further improve the intra-session protocol.

8 Conclusion

Cooperation and autonomy are notable properties of loosely coupled distributed applications in which autonomous processes cooperate with each other to exchange and share information beyond administrative boundaries. This paper proposed a novel substrate for cooperative autonomous systems, one that integrates remote procedure calls and virtual memory management. The distinctive feature of this substrate, the VM-integrated RPC, is that it combines the advantages of the remote procedure call (RPC) with the advantages of distributed shared memory (DSM). As conventional RPCs do, it preserves the autonomy of processes since it requires cooperating processes to agree only on logical interfaces. In addition, like DSM systems, it facilitates the cooperation of distributed processes since it enables any data types, especially references, to be shared beyond address space boundaries. Compared with conventional RPCs, the VM-integrated RPC is an extension to deal with remote references. And compared with conventional DSMs, it is an extension to preserve the autonomy and for application in heterogeneous environments.

The VM-integrated RPC provides distribution transparency at the instruction-code level, and this means that the instruction code that manipulates virtual addresses can manipulate remote references without any modifications. This allows all references, either local or remote, to be uniformly and efficiently represented by virtual addresses. More significantly, the instruction-code-level transparency does not assume any specific language implementations. Thus the VM-integrated RPC is essentially applicable to any programming language. Furthermore, this transparency enhances the reusability of existing compilers and libraries and does not restrict any compiler optimizations or compilation techniques.

Our execution model assures the shared semantics of shared data, but some situations would require the *copy* semantics: when a reference is passed, the entire closure referred to by that reference is copied and given an identity different from that of the original closure. If the entire closure is always replicated onto a target remote address space, incorporating the copy semantics into our mechanism is straightforward, since we have only to assign a new UID for every replicated data. If data are replicated in a lazy manner (i.e, copy-on-reference), however, the image of the closure at the reference passing time may be lost because the not-yet-replicated portion of the closure can be updated. We speculate that application programmers will be able to control the "degree" of the copy semantics using pickling modules if the system provides an option that directs the system to assign a new UID to replicated data.

In cooperative autonomous systems, the underlying system might be required to provide the mechanism for fault tolerance. Our current implementation does not address fault tolerance, but the transaction-like mechanism provided by the inter-session protocol could support fault tolerance. Another important issue of cooperative autonomous systems is security. This is still an open problem and is an active area of research. It will be interesting to develop mobile object systems based on our technology. For example, if we combine our technology with mobile code techniques such as Java byte code [17] and OmniCode [1], the combination would enable native code objects to roam over the world in order to accomplish a given task.

References

[1] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proc. of ACM Conf. on Programming Language Design and Implementations*, pages 127–136, May 1996.

[2] J. S. Auerbach, A. S. Gopal, M. T. Kennedy, and J. R. Russel. Concert/C: Supporting distributed programming with language extensions and a portable multiprotocol runtime. In *Proc. IEEE 14th Int. Conf. on Distributed Computing Systems*, pages 152–159, 1994.

[3] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[4] Brian N. Bershad et al. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

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

[6] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *14th ACM Symposium on Operating Systems Principles*, pages 217–230, December 1993.

[7] Vinny Cahill, Sean Baker, Chris Horn, and Gradimir Starovic. The Amadeus GRT - generic runtime support for distributed persistent programming. In *OOPSLA Proceedings 1993*, pages 144–161. ACM, 1993.

[8] Luca Cardelli. A language with distributed scope. In *Proc. ACM Symp. on Principles of Programming Languages*, January 1995.

[9] R. G. G. Cattell. *Object Data Management — Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, 1991.

[10] David Chappell. *Active X and OLE.* Microsoft Press, 1996.

[11] Randy Chow and Theodore Johnson. *Distributed Operating Systems & Algorithms.* Addison-Wesley, 1997.

[12] M. Mira da Silva, M. Atokinson, and A. P. Black. Semantics for parameter passing in a type-complete persistent RPC. In *Proc. IEEE 16th Int. Conf. on Distributed Computing Systems*, pages 411–418, 1996.

[13] P. J. Denning. The working set model for program behavior. *Commun. of the ACM*, 11(5):323–333, May 1968.

[14] R. Draves, M. Jones, and M. Thompson. MIG — the Mach interface generator. Technical report, Carnegie-Mellon University, February 1988.

[15] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, 1989.

[16] Phillip B. Gibbons. A stub generator for multilanguage rpc in heterogeneous environments. *IEEE Transactions on Software Engineering*, SE-13(1):77–87, January 1987.

[17] J. Gosling and H. McGilton. The Java language environments: A White Paper. Technical report, Sun Microsystems, 1995.

[18] Graham Hamilton, Michael L. Powell, and James G. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79, December 1993.

[19] Roger Hayes and Richard D. Schlichting. Faciliating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering*, SE-13(12):1254–1264, December 1987.

[20] Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 106–119, December 1993.

[21] Java Soft. *Java Remote Method Invocation Specification*, 1997. available from http://www.javasoft.com/.

[22] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[23] K. Kato, A. Ohori, T. Murakami, and T. Masuda. Distributed C language based on a higher-order remote procedure call technique. In *Advances in Software Science and Technology*, volume 5, pages 119–143. Academic Press, 1993.

[24] K. Kono, K. Kato, and T. Masuda. Smart remote procedure calls: Transparent treatment of remote pointers. In *Proc. IEEE 14th Int. Conf. on Distributed Computing Systems*, pages 142–151, 1994.

[25] K. Kono, K. Kato, and T. Masuda. An implementation method of migratable distributed objects using an RPC technique integrated with virtual memory management. In *Proc. of European Conf. on Object-Oriented Programming (ECOOP), LNCS 1098*, pages 295–315. Springer-Verlag, 1996.

[26] K. Kono, K. Kato, and T. Masuda. Comparison and performance study of parameter-passing semantics and implementations of remote method invocation. In *Proc. of Second IASTED European Conference on Parallel and Distributed Systems (Euro-PDS'98)*, pages 286–292, July 1998.

[27] Richard T. Kouzes, James D. Myers, and William A. Wulf. Collaboratories: Doing science on the Internet. *IEEE Computer*, 29(8):40–46, August 1996.

[28] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[29] J. E. B. Moss. Working with persistent objects: to swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[30] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[31] Object Management Group. The Comon Object Request Broker: Architecture and specification. Technical report, Object Management Group, December 1991.

[32] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *Proc. 20th ACM Symp. on Principles of Programming Languages*, pages 99–112, January 1993.

[33] Open Software Foundation. OSF DCE release 1.0 developer's kit documentation set, February 1991.

[34] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In *European Conf. on Object-Oriented Programming (ECOOP)*, pages 191–204, 1989.

[35] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of Fifth Int'l Workshop on Persistent Object Systems*, pages 11–33, 1992.

[36] Alexander D. Stoyenko. SUPRA-RPC: SUbprogram PaRAmeters in remote procedure calls. *Software Practice and Experience*, 24(1):27–49, January 1994.

[37] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64, May 1988.

[38] P. Wilson. Pointer swizzling at page fault time: efficiently supporting huge address spaces on standard hardware. *ACM Computer Architecture News*, pages 6–13, June 1991.

[39] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proc. of Int'l Workshop on Object Orientation in Operating Systems*, pages 364–377. IEEE, 1992.

[40] XEROX Corporation. *Courier: the Remote Procedure Call Protocol*, Dec. 1981. XEROX system integration standard XSIS-038112.

[41] S. Zhou, S. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554, September 1992.