An Implementation Method of Migratable Distributed Objects Using an RPC Technique Integrated with Virtual Memory Management

Kenji KONO[1]* and Kazuhiko KATO[2] and Takashi MASUDA[1]

[1] Department of Information Science, Graduate School of Science,
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
Email: {kono,masuda}@is.s.u-tokyo.ac.jp
[2] Institute of Information Sciences and Electronics, University of Tsukuba
Tsukuba, Ibaraki 305, Japan
Email: kato@is.tsukuba.ac.jp

Abstract. Object abstraction is indispensable to construction of distributed applications to encapsulate the details of execution entities. By applying an RPC technology integrated with virtual memory management, this paper presents a novel approach to implementing migratable distributed objects. The novelties of the approach are transparency achieved at the instruction code level, distributed dynamic methods, and applicability to heterogeneous environments. The instruction code level transparency naturally accomplishes object migration and enables efficient manipulation of migrated objects. The distributed dynamic methods provide the programmers with flexible control of activities.

1 Introduction

Construction of distributed application, which consists of autonomous execution entities running on loosely coupled machines, is hindered by the lack of knowledge on remote execution entities. Thus, it is essential to abstract these entities as the interacting modules with well-defined interfaces and encapsulate their details from the others. This observation naturally leads us to building an application as a collection of *distributed objects*, which are the direct embodiment of those modules. Research efforts in this past decade have invented various implementation mechanisms for object distribution. The mechanisms proposed so far can be classified into three approaches: 1) the virtual machine approach, 2) the proxy-object approach, and 3) the distributed shared memory (DSM) approach.

The virtual machine approach prepares a virtual machine [11, 18, 9, 22], often implemented as a byte-code interpreter, that provides a transparent mechanism for passing messages among distributed objects. Since the virtual machine encapsulates the underlying hardware and operating systems, this approach provides

* Research Fellow of the Japan Society for the Promotion of Science.

a platform on top of which we can design and implement distributed object-oriented programming languages with a high degree of distribution transparency. In addition, *object migration* is achieved with relative ease because of the encapsulation by the virtual machine. Using object migration, application programmers might improve the execution performance by gathering the objects on one address space among which messages are frequently exchanged. However, the virtual machine approach pays a high price for these advantages; it gives up the effective program execution in native machine code.

In the proxy-object approach, a remote object is invoked indirectly through invocation of the *proxy object* [19] that is a counterpart of the *stub* routine in a remote procedure call (RPC) [3]. A proxy encapsulates communication details from application programmers. In this approach, a language preprocessor emits the code of the necessary proxies and replaces remote object invocations with the corresponding local proxy calls, thereby accomplishing the literal level transparency. Given a language preprocessor, we can use the conventional native code compilers *not* designed for the distribution purpose. The most attractive feature of the proxy approach is flexibility; various mechanisms can be encapsulated in proxies. For example, proxy-based systems become applicable to heterogeneous environments if the code of data conversion is embedded in proxies. Object migration can be achieved with some restrictions on the optimizations by compilers. This issue is discussed later in Section 3.5.

The DSM approach is to build object systems on top of DSMs implemented by either hardware [15] or software [16]. Once DSM is provided, distributed object systems are not difficult to implement, since the DSM layer provides a single logical address space shared among distributed sites. Clouds [6] takes this approach. Most DSM implementations use the memory management unit (MMU) hardware to trap an access to absent pages, and cache the accessed pages on the local memory. Object systems built on those implementations bring about an effect similar to object migration and achieve transparency at the instruction code level. Thus, existing compilers can be used with few modifications. Generally, DSMs assume homogeneous environments because it is fairly difficult to implement them in heterogeneous environments. Even if implemented in heterogeneous environments, these systems are subject to many restrictions [23].

To summarize, the virtual machine approach provides a platform suitable for object migration, but precludes execution of application programs in native machine code. The proxy-object and the DSM approaches allow efficient execution in native machine code. The former is applicable to heterogeneous environments, but restricts compiler optimizations if object migration is achieved. Although the latter is not applicable to heterogeneous environments, it automatically achieves object caching similar to object migration.

Recently, a new RPC technology has been proposed [14, 13] that enables quite transparent treatment of remote pointers by integrating the RPC and virtual memory management techniques without sacrificing the virtues of RPCs, such as their applicability to heterogeneous environments. In this paper we describe an approach to implementing a system for migratable distributed objects by applying the proposed RPC technology. The implemented system has been named TRAP-DO. The notable feature of TRAP-DO is to put together the two advantageous points of the above-mentioned conventional approaches that enable native code execution: one advantageous point is transparency at the instruction code level in the DSM approach and the other is flexibility in the proxy-object approach.

In the TRAP-DO system, object references have a uniform representation; that is, remote objects are referenced by virtual addresses directly accessible by ordinary CPUs in the same manner as a local object reference. An attempt to access an absent (remote) object is detected by the hardware for virtual memory management. Thereafter, the accessed object migrates to the local address space. After the migration, the access to the migrated object is completely the same as the access to local objects. Thus, application programs can execute in native machine code, and no restrictions are imposed on compiler optimizations. This mechanism seems similar to the DSM approach; both the DSM approach and the TRAP-DO approach accomplish the instruction code level transparency and allows remote objects to be referenced by virtual addresses. However, the TRAP-DO approach is quite different from the DSM approach. Unlike the DSM approach, TRAP-DO does *not* share a single logical address space. Each address space manages its own local memory and reconstructs memory image of the migrated object independently of other address spaces. Thus, the TRAP-DO approach is applicable to heterogeneous environments if the internal representation of the migrating object is converted according to the target machine architecture to preserve the logical type of the migrated object. In addition, TRAP-DO is applicable to a variety of existing programming languages. This property contrasts with typical systems in which either original languages [11, 1] are developed or a single language is extended for distribution [20, 2, 12].

Generally, it is a difficult task to migrate objects in existing systems as pointed out in Douglis and Ousterhout [8]. In brief, this is because the entire state of an object may be scattered in the operating system data structures. TRAP-DO provides the *distributed dynamic methods* that utilizes the aspect that RPC can be applied to dynamic linking, as discussed in Hayes *et al* [10]. This mechanism bypasses the difficulties of migration. For example, consider the case where an object has a pointer to the data structure held by the operating system, and a method of the object interacts with the operating system using the pointer. If this object migrates to another address space and that method is executed there, the result would be an unexpected one. With a distributed dynamic method, the programmer can specify an address space on which a method is executed. Using this mechanism, in the above example, the programmer has only to direct the runtime system to execute the method in question at the appropriate address space. This mechanism also allows the programmers to utilize diverse architectures in a heterogeneous environment. For instance, they can specify a particular implementation to be executed on the machine with special equipment.

The rest of the paper is organized as follows. Section 2 overviews the entire system. The implementation is described in Section 3. Section 4 reports the experimental results. Section 5 concludes the paper.

2 System Overview

2.1 Basic Concepts

TRAP-DO supports from fine to medium-grain objects; a single address space can hold many objects at once. A *class* is a template from which objects are created, and every object is an instance of some class. TRAP-DO provides *passive objects* wherein the threads and objects are completely separate entities; a thread is not bound to a single object. We refer to a chain of nested invocations of methods as an *action*, and a distributed thread of control as an *activity*. In the passive object model, a single activity executes all the methods associated with an action, migrating from one object to another.

TRAP-DO provides two kinds of objects: *global* and *local* objects. The references to global objects can be passed beyond the address space boundaries. Thus, global objects can be invoked from outer address spaces, and may migrate to other address spaces. On the other hand, the references to local objects can not be transferred to remote address spaces. If an attempt is made to pass a reference to a local object to a remote address space, TRAP-DO automatically detects it and raises an exception. The programmers specify global or local for each object at the instantiation time.

In TRAP-DO, when an activity invokes a method, the target object migrates to the address space where the invoking activity is running, and the method is executed at that address space. This mechanism is completely hidden from the programmers; they need not to be aware of the location of objects. In this execution model, an activity is always bound to one address space. To allow the programmers to flexibly control the location of *activities*, TRAP-DO provides the mechanism called the *distributed dynamic methods*. When invoking a method, the programmers can *dynamically* specify the address space on which the method should be executed, regardless of the location of the target object. Using this mechanism, the programmers can incorporate various strategies into their applications; for example, they can distribute the computation for load sharing, or can exploit the heterogeneity of the distributed environment by executing a specific method on a machine with special equipment. Of course, location-dependent methods can be executed at the appropriate address space.

The current implementation of TRAP-DO employs a string composed of the host name and the service name as a logical name of an address space. For instance, a string "*Mozart:address-book*" specifies an address space that provides the service *address-book* on the host *Mozart*. The host name can be omitted so that the programmers can simply specify the service name. In this case, one host that provides the specified service is selected by the runtime system. The logical name string can be generated by programmers at runtime. From a logical name, the runtime system determines the actual address space and makes the

```
 1: //class definition
 2: class X {
 3:     //instance variables
 4: public:
 5:     void foo(void);
 6:     void bar(void)@"host:service";
 7: };
 8:
 9: ...
10: {
11:     TransactionalSession(hos, hls){
12:         X* o = new (global)X;   //instantiate a global object
13:
14:         //methods are executed on the default address spaces
15:         o->foo();        //executed locally
16:         o->bar();        //executed at "host:service"
17:
18:         //changing the executing address space
19:         o->foo()@"host:service"; //executed at "host:service"
20:         o->bar()@"local";        //executed locally
21:
22:         //an example of load sharing
23:             //dynamically select an address space
24:         const char* address_space = LoadSharing();
25:             //then execute foo() at the address space
26:         o->foo()@address_space;
27:     }
28: }
```

Fig. 1. An Example Code of TRAP-DoC.

activity migrate to the specified address space. The format of a logical name string and how it is interpreted are the problem of naming and beyond the range of this paper. TRAP-DO does not preclude more elaborate logical names; as a future extension, we are planning to employ a variant of URL (uniform resource location).

2.2 Language and Example

As a user programming language, TRAP-DO currently provides TRAP-DoC that extends ANSI C and provides C++-like notation to deal with distributed objects. In order to concentrate on the distribution issues, the current TRAP-DoC does not support inheritance.

Figure 1 shows an example of a class definition. This example defines a class `X` that provides two methods `foo()` and `bar()`. While the declaration of `foo()` is the same as that in C++, the declaration of `bar()` is annotated with `"host:service"`. This annotation defines the default address space on which this method is executed. In this example, `bar()` is executed by default on the address space specified by `host:service`. If this annotation is omitted like the declaration of `foo()`, the default address space is the local address space. At the lines 15 and 16 in Fig. 1, the methods are executed at the default address spaces. The default address spaces can be dynamically changed as shown in the

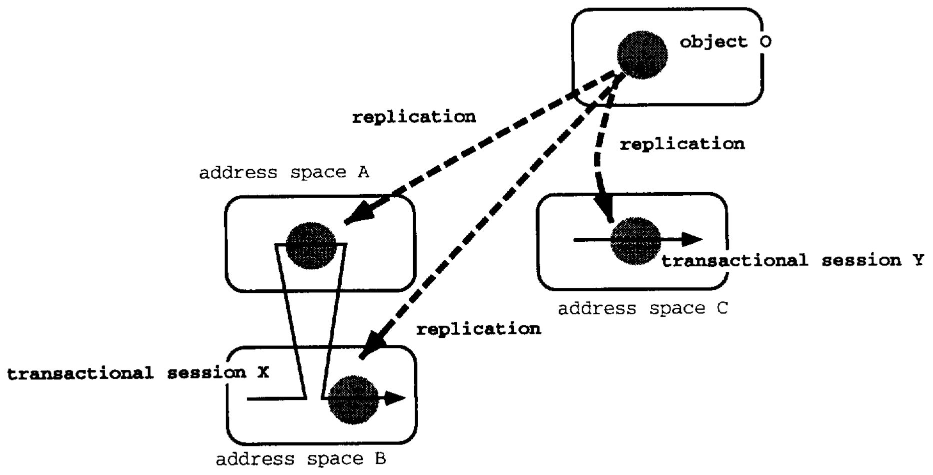


Fig. 2. Transactional session. Two transactional sessions X and Y are operating on the object O. The transactional session X operates on two replicas of the object O.

lines 19 and 20. The string "`local`" is an abbreviation of the logical name of the local address space. At the line 26, the executing address space is dynamically changed for load sharing. The function `LoadSharing()` returns the logical name of the host whose load is the lightest, and then the `foo()` is executed there. Note that TRAP-DOC accepts C++ programs without modifying the semantics of method calls, except that TRAP-DOC does not support inheritance.

2.3 Transactional Session

Since TRAP-DO allows multiple activities to execute concurrently and to perform update operations on objects, some problems must be dealt with, including maintaining the coherent state between replicas and synchronizing multiple activities of actions. To attack these problems we introduce *transactional session* into the system. A transactional session serves as a *transaction* that serializes the operations performed by multiple activities, and also serves as a *session* that determines the period during which the references to remote objects are valid. The references to remote objects are not valid beyond the transactional session; the programmers must split their programs into several transactional sessions among which remote object references are not shared. A programmer declares the beginning and the end of a transactional session. In Fig. 1 a transactional session is declared at the line 11. The role of the parameters "`hos`" and "`hls`" is explained in Section 3.2.

A set of operations performed within a transactional session is guaranteed to be atomic, serializable and isolated from the others by the *inter-session* protocol. In Figure 2, two activities within transactional sessions X and Y are simultaneously performing update operations on the replicas of the object O. The inter-session protocol serializes the transactional sessions X and Y. As shown in this figure, there may exist multiple replicas of a single object within a transactional session. Thus, the coherency between these replicas must be maintained. In this figure, the object O is replicated on the address spaces A and B within the transactional session X. The protocol named *intra-session* protocol maintains the coherency between these replicas. This protocol guarantees one-copy semantics within a transactional session. In other words, the activity always observes the results of the latest update within a transactional session. In TRAP-DO, the one-copy semantics is relatively easy to implement, since the synchronous property of method invocations assures that there is a single activity in each transactional session. This property simplifies the protocol and makes it efficient.

Management of replicas causes the problem of when to release them in addition to the problem of the coherency between them. We adopt the *per-session replication* policy for isolation of transactional sessions; replicas are not shared between transactional sessions. By using the per-session replication policy, we can dispense with distributed garbage collection to reclaim the replicas. Since object migration may occur only during a transactional session, replicas of objects are in use only within the transactional session. The system can simply dispose of the replicas created during a transactional session when it reaches the end of the transactional session, since the replicas are not shared by other transactional sessions.

3 Implementation

Our migration mechanism consists of three parts. Section 3.1 explains the integration of virtual memory management and object migration. Section 3.2 describes the intra-session protocol for replica coherency. Section 3.3 describes the inter-session protocol for synchronization. Section 3.4 describes the support from the programming language layer, and explains the mechanism of the distributed dynamic methods and stub generation. Section 3.5 discusses some aspects of TRAP-DO.

3.1 Integrating Virtual Memory Management with Object Migration

To provide transparent and efficient references to objects, TRAP-DO enables objects to be referenced by virtual addresses regardless of the location of the objects. Since a virtual address becomes meaningless outside the address space where it is defined, some mechanism must be provided to preserve the logical links between objects even when virtual addresses are transferred to remote address spaces. TRAP-DO introduces *universal object identifiers (UIDs)* valid in the entire distributed system. The UID of an object consists of three parts: 1) the address space identifier of the object's birthplace (typically a pair of the site ID and the process ID in the site), 2) the address of the object in the birthplace, and 3) the specifier of the class to which the object belongs. We assume that the

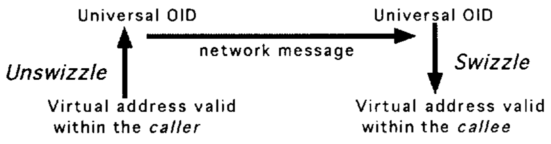

Universal OID
network message
Universal OID
Unswizzle
Swizzle
Virtual address valid within the *caller*
Virtual address valid within the *callee*

Fig. 3. Swizzling and unswizzling of a UID.

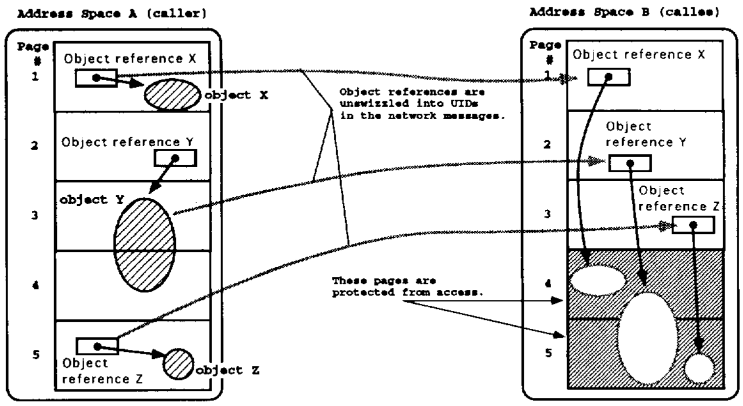
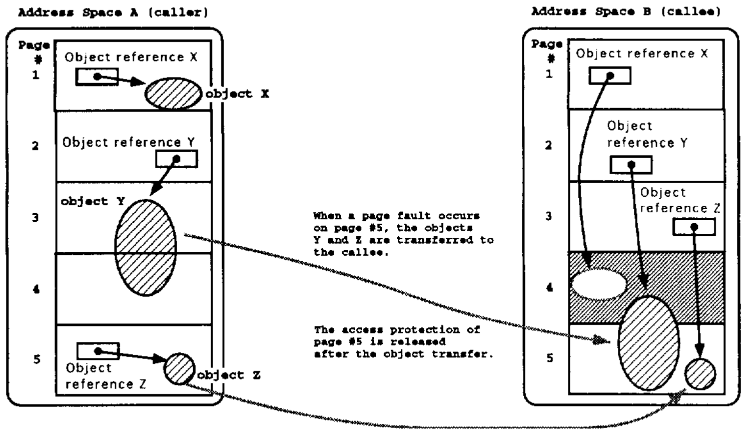


Fig. 4. Just after three object references are swizzled in the address space Y.

system can obtain an actual data structure from a class specifier by querying a database that serves as a network name server.

When an attempt is made to pass a reference (virtual address) to a remote address space as an input or output argument of a remote method, it is translated into the UID at the caller, and the UID is translated into a virtual address at the callee side. The translation from UID into a virtual address is called *pointer swizzling* and the reverse translation is called *pointer unswizzling* (see Fig. 3). When the callee receives the UID, it is not straightforward to swizzle the UID into a virtual address, since the object referenced by the UID exists only at the remote address space at this time. To swizzle the UID into a virtual address, the callee allocates a *protected* area in a memory page or pages that MMU protects from read and write access. The size of the allocated area is the same as the object referenced by the UID, the size of which is obtained from the class specifier embedded in the UID. The transferred UID is swizzled into the address of the allocated area. Note that the allocated area is empty at this time. The actual object *will* be copied to the location later when necessary. Figure 4 illustrates this situation. Three object references X, Y and Z are transferred from the caller to the callee. On the callee side, three empty areas are allocated respectively for each object in the two protected pages. As shown in the figure, one protected page can contain a set of remote objects, and an object can cross page boundaries.

MMU detects the first attempt to access an object allocated in a protected



Address Space A (caller)
Address Space B (callee)
Page #
1
2
3
4
5
Object reference X
object X
Object reference Y
object Y
Object reference Z
object Z
When a page fault occurs on page #5, the objects Y and Z are transferred to the callee.
The access protection of page #5 is released after the object transfer.

Fig. 5. When a protected page is accessed, all objects in the page are transferred.

page, and raises an access-violation exception. The operating system kernel is informed *a priori* that the runtime system handles this exception. When an exception is raised, the exception handler determines at which location this exception was raised. At this point, the accessed object is transferred from the caller to the callee. All objects allocated in the same page must be transferred at this time, because the access to the page can not be detected after the release of the page protection. When the objects are transferred, their internal representations are encoded and decoded to preserve their types in a heterogeneous environment. We can use the standard methods except in the case of references. They must be unswizzled and swizzled if embedded in the transferred objects. After the objects are transferred, the operating system kernel is directed to release the access protection of the page. Then, the faulting activity is resumed. At this time, only the read access is permitted to detect write access afterwards.

When an object migrates to a remote address space, TRAP-DO replicates the object and caches the replica to the remote address space. Since only one activity exists within a session, there is an "up-to-date" replica. We call the up-to-date replica *hot object*, and *hot location* the identifier of the address space from which the hot object is available. The runtime system maintains an *object allocation table*, the entries of which are the page number, the offset within the page, the UID, and the hot location. The hot location indicates the address space from which the hot object is available. The intra-session protocol manages the hot locations to maintain the replica coherency. The runtime system refers to this table at the page fault time to determine which object is to be transferred from which address space. This table also serves to preserve logical links between objects when remote invocations are nested. Whenever required, a reference can be translated into and from a UID by this table. In the example shown in Fig. 4, the object allocation table would be like Table 1. Figure 5 shows the transfer of objects in the example shown in Fig. 4. In Fig. 5 a page fault is detected on page

#5. By looking up the object allocation table, the runtime system decides the object Y crosses the page boundary, since the offset$_Y$ plus the size of the object Y is greater than the page size. Then the objects Y and Z are transferred from the address space A.

page #	offset	UID	Hot location
4	offset$_X$	X	A
4	offset$_Y$	Y	A
5	offset$_Z$	Z	A

Table 1. Object allocation table.

In the description above, an object is transferred on demand. If objects are traversed following the references, a terrible situation arises. The number of page faults and network communications are both increased. If fine-grain objects are transferred, network bandwidth is not utilized well. To avoid this situation, when passing a reference or fetching an object, TRAP-DO transfers a set of objects: a certain depth of the *transitive closure* of the passed object. There are many alternative algorithms to take a certain depth of transitive closure. By default, TRAP-DO uses the breadth-first traversal algorithm, and the programmers can explicitly specify the *closure size* parameter, the maximum amount of the transferred objects. At the traversal time, the system detects cycles, and prevents page faults by examining the page protection mode before accessing a page.

3.2 Replica Coherency Protocol

Generally, all replication techniques inherently require a protocol for replica coherency. TRAP-DO guarantees one-copy semantics of replicas within a transactional session by the *intra-session* protocol. Since each transactional session is completely isolated from the others by the per-session replication policy described in Section 3.3, we can assume there is a single activity in the system when discussing the intra-session protocol. Thus, it is sufficient to provide the activated address space with the up-to-date state of objects. The basic strategy of the protocol is to transfer a *hot location set* to the activated address space, together with the arguments of the method. As mentioned in Section 3.1, the intra-session protocol manages the hot location column of the object allocation table. The hot location set is a collection of a pair of the UID and the hot location of the modified object, and keeps track of the hot locations of the modified objects. When the activity attempts to access an out-of-date object, MMU detects it and the runtime system performs an RPC to obtain the hot object from the hot location. If hot objects are directly transferred to the activated address space instead of hot locations, we can expect better performance if there is locality of reference. In the intra-session protocol, the *hot object set*, a collection

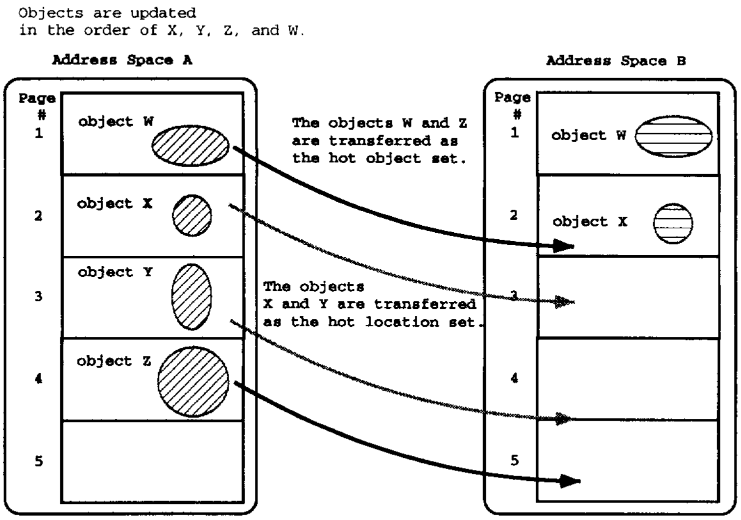

Objects are updated
in the order of X, Y, Z, and W.
Address Space A
Address Space B
Page #
1
2
3
4
5
object W
object X
object Y
object Z
The objects W and Z are transferred as the hot object set.
The objects X and Y are transferred as the hot location set.

Fig. 6. Transfer of the hot object and location sets. The objects W and Z are transferred as the hot object set, and the others are transferred as the hot location set.

of the hot objects, is directly transferred to the activated address space for the *recently* modified objects. For the *less recently* modified objects, the hot location set is transferred. The programmer can specify the *HOS* and *HLS* parameters, which limit the sizes of the hot object set and the hot location set, respectively.

TRAP-DO implements the protocol described above in the following way. The runtime system maintains a link of dirty (modified) pages. When MMU detects an update on a page, the runtime system determines the page to which the update is applied, and links the entry of the page to the dirty page link in LIFO order. We can use this link to approximately determine the order of modifications; a recently modified page comes first in the link. When the activity attempts to migrate to another address space, the dirty page link is traversed to create a hot object set. Traversing the link, the objects on the modified pages are marshalled into the hot object set, until the size of the hot object set reaches the HOS parameter. After the creation of the hot object set, the hot location set is created for the modified objects not marshalled into the hot object set. Then the hot object and location sets are transferred, together with the arguments of the remote method. Figure 6 shows this situation. On the address space A, four objects W, X, Y, and Z are updated in the sequence of X, Y, Z, and W. Thus, the pages #1, #4, #3, #2 are linked to the dirty page link in this order. In this example, the objects W and Z are marshalled into the hot object set, and the others are transferred as the hot location set. On the address space B, the objects W and X already exist; they are out-of-date now.

The hot object and location sets are properly reflected to the objects on the receiving side. The objects in the hot object set are replicated into the receiving address space in the same way as described in Section 3.1, since this replication

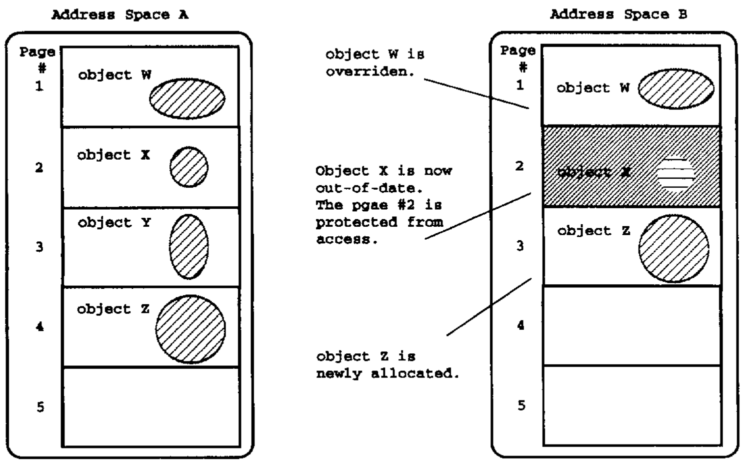


Fig. 7. Reflecting the hot object and location sets to the address space B.

can be regarded as object migration. The object whose UID is in the hot location set is out-of-date. The runtime system looks up the object allocation table for each UID in the hot location set, and closes all access permissions of the pages on which the out-of-date objects reside. The hot location column of the object allocation table is also updated to the new hot locations. When an attempt is made to access an out-of-date object, a page fault occurs, and the hot objects can be migrated as described in Section 3.1. Figure 7 illustrates this situation. The objects W and Z in the hot object set are replicated into the address space B. The transferred W overrides the object W on B, and the object Z is newly allocated. The runtime system determines the object X is out-of-date from the hot location set, and invalidates the page on which it resides.

When the size of the hot location set exceeds the HLS parameter, all hot objects are written back to their original locations, and the replicas created during the transactional session are invalidated. Thereafter, the hot location and object sets are reset to zero in size. We can always obtain the hot objects from their original locations until the activity moves beyond the address space boundary. The intra-session protocol works properly even when remote invocations are nested. While activated, the runtime system maintains the hot location set and the UIDs of the objects in the hot object set, in order to keep track of all the modifications during the session. The memory area for keeping them does not suppress the application programs, since the HOS and HLS parameters limit the sizes of the hot object and location sets.

3.3 Concurrency Control Protocol

Since multiple transactional sessions execute concurrently, concurrency must be controlled by the system. In TRAP-DO, the *inter-session* protocol guarantees serializable execution of transactional sessions. The inter-session protocol employs an optimistic approach. Generally, optimistic approaches have the following attractive features as against pessimistic approaches. The pessimistic approaches require additional network communications to acquire and release locks or to operate on distributed semaphores while the transaction is being executed. In addition, the problem of distributed deadlock must be dealt with. As our system assumes fine-to-medium grain objects, deadlocks are expected to be caused frequently. On the other hand, optimistic approaches enhance the effect of object caching, because they require no additional communications during the execution of a transactional session. Moreover, they do not cause deadlocks. The major drawback of the optimistic approaches is that they may cause cascading roll backs. Our current protocol avoids them by complete isolation of transactional sessions.

TRAP-DO manages a heap area based on the *per-session replication* policy to isolate a transactional session from the others. This section outlines the mechanism of the per-session replication and the inter-session protocol. The details are presented in our paper in preparation [13]. TRAP-DO divides a heap into three distinct parts: pages for replicas, global objects, and local objects. Since global and replicated objects may be shared among multiple activities, TRAP-DO maintains the coherent state of those objects. The programmers can avoid concurrency control cost by allocating objects in local objects' pages. Since local objects are not referenced from outer address spaces, TRAP-DO need not control the concurrency. TRAP-DO can detect an attempt to pass a reference to a local object to a remote address space, because a reference to a local object cannot be unswizzled into the UID by the object allocation table mentioned in Section 3.1; the entry is not found. Each transactional session prepares its own object allocation table. Thus, replicas are not shared among transactional sessions; transient states of replicas are automatically isolated from the other transactional sessions. To isolate updates on the global object pages, TRAP-DO creates a session-local image of the global pages on a copy-on-write basis. When an activity attempts an update on a global object page, the coherent image of the page is preserved, and then the update is applied directly to the global object page. When the activity switches, the transient image is preserved for each session, and the consistent image is reloaded. Consequently, TRAP-DO prevents each transactional session from observing the transient state of the objects in other transactional sessions.

In the optimistic concurrency controls, the updates performed during a transactional session are checked at the validation phase to ensure that the transactional session does not violate the serializability. In the current inter-session protocol, the transactional session that reaches its end fastest is successfully validated. If a transactional session is validated successfully, the hot objects are written back to their original global pages, and the replicas are disposed of simply by releasing the replica pages of the transactional session. If a transactional session fails to be validated, it is rolled back. The roll back is done by releasing the replica pages and transient images of the global pages.

3.4 Distributed Dynamic Methods and Stub Generation

The runtime system of TRAP-DO described in previous sections cooperates with the programming language layer. This language layer performs three tasks: attaining distributed dynamic methods, stub emission, and generation of closure traversal routines. The current user language, TRAP-DOC, is a preprocessor from TRAP-DOC to C.

Generally, distributed dynamic methods are invoked through proxies within which the target address space is determined by interpreting the specified logical name of the address space. However, in TRAP-DOC, most method invocations can be statically determined to be local calls. For example, invocations of methods, both the declaration and the invocation of which are annotated with nothing, are local calls and can be expanded into ordinary procedure calls by the preprocessor. If the target address space is not determined statically, the preprocessor emits proxies, or stubs, for the remote method invocation. This proxy interprets the logical name of the address space and performs an RPC to the target address space. As pointed out in Section 2.1, the mechanism of naming is independent of the TRAP-DO's runtime mechanism described in previous sections and various naming policies can be implemented on top of TRAP-DO. The proxy generation technique used in TRAP-DO allows higher order functions to be passed among distinct heterogeneous address spaces. This stub generation technique is based on the one described in Ohori and Kato [17].

As described in Section 3.1, when passing a reference to an object to a remote address space, a certain depth of the transitive closure of the reference is transferred. The language preprocessor generates a *traversal routine* that creates this closure of the passed reference. By default, TRAP-DOC emits the traversal routine for breadth-first traversal. Although the traversal mechanism is hidden from the programmers, they can incorporate their own algorithms into the system. TRAP-DO provides the interface between the runtime system and the programmers. If the programmers' traversal routines conform to this interface, the system detects the cycles and prevents page-faults at the traversing time. This interface and example codes are shown in another paper [13].

3.5 Discussion

To highlight some aspects of TRAP-DO, we contrast the object migration mechanism based on proxy objects with that of TRAP-DO. The proxy object approach is applicable to object migration by encapsulating the mechanism of migration within proxies. Many systems using proxies for migration examine with software whether the target object is present or not, each time before using a reference. In this mechanism, each access to an object requires additional software overhead. Amadeus [4] proposed a mechanism to avoid this additional overhead. It uses a proxy object called "O proxy" to trap attempts to access absent objects. O proxy contains no data but is the same size as the object it represents. The code bound to the proxy implements the same interface as the absent object. When a method of the O proxy is invoked, it notifies the Amadeus runtime system of the attempted invocation. Then the runtime system overlays the O proxy with the real object, and forwards the invocation to the real object. This approach does not work properly unless all method invocations are implemented as procedure calls even at the instruction code level. Hence, in the Amadeus approach, compilers can not use some sorts of optimization techniques [7, 5] such as inline-expansion that compilers of object-oriented programming languages use. On the other hand, TRAP-DO does not impose any restrictions on the code generation by compilers, because it accomplishes transparency at the instruction code level. Thus, in TRAP-DO, compilers can optimize frequent message exchanges among inter-related objects, and application programs can run as efficiently as possible after object migration.

The runtime behavior of TRAP-DO is adjustable to the access pattern of applications. In TRAP-DO, the programmers can explicitly specify the granularity of migration by the closure size parameter, which limits the size of the transferred transitive closures. For example, suppose that the reference which designates the head of a linked list is passed, and the linked list is traversed on the local address space. If the closure size parameter is set to zero, the linked objects are transferred on demand. Hence, the number of page faults and network communications increase, but the amount of the transferred data is minimized, since just the accessed portion of the linked list is transferred. If the closure size parameter is set to infinity, no page fault is incurred and the network communication is required only once. However, since all the linked objects are transferred at once, unnecessary objects are also transferred that are not accessed in the local address space. According to the access pattern of the applications, the programmers can specify the closure size parameter to tailor the runtime behavior of the system to the applications.

4 Experimental Results

In this section, we measure the basic performance of TRAP-DO and validate the discussion in Section 3.5. The prototype TRAP-DO is implemented on SunOS 4.1.3 running on SUN Sparc workstations (SuperSPARC, 60MHz, SPECint92 98.2, SPECfp92 107.2). They have 96 Mbytes of main memory and are connected by a 10 Mbps Ethernet network. The current implementation uses the TCP/IP network protocol and specifies the `TCP_NODELAY` option in the socket system call so that small packets are sent without buffering. As a canonical data representation, TRAP-DO uses the XDR (eXternal Data Representation) [21] that guarantees the transformation of basic data types such as integers, floating point numbers, and strings between CPUs with different architectures. Although the current system consists of only a few SPARC stations, the system is carefully designed to deal with heterogeneity. Therefore, the experimental results reflect heterogeneity overheads such as the data representation overhead.

4.1 Basic Performance

As described in Section 3.1, the TRAP-DO runtime system handles a page fault before migrating an object. The experimental results in this section show that the cost of page fault handling is small compared with the cost of network communication. In this experiment, objects each of which is 64 bytes in size are migrated from a remote address space to the local address space. This migration is executed in the following steps. First, a page fault is detected. Second, the exception handler requests the remote address space to transfer the objects. After the object migration, activity is resumed. Varying the number of migrated objects, we measured the total time of the migration and the cost of the page fault handling. Table 2 shows the result. In this result, the proportion of the cost of page fault handling to the total cost is from 2% to 21%. However, since the TRAP-DO runtime system transfers all objects allocated in a page at once, the size of transferred objects may be in the order of 4 Kbytes or 8 Kbytes. Thus, the proportion of the cost of page fault handling is considered to be from 2% to 4%.

Total size of objects (bytes)	64	256	1024	4096	8192
Cost of migration (ms)	1.4	1.7	3.2	7.3	12.7
Cost of page fault handling (ms)	0.3	0.3	0.3	0.3	0.3
page fault handling / migration (%)	21	18	9	4	2

Table 2. Proportion of the cost of page fault handling to the cost of network communication.

4.2 Closure Size

The closure size parameter plays an important role in TRAP-DO as discussed in Section 3.5. This section examines the effect of the closure size parameter. The experimental subject was a traversal of a complete binary tree. Each node of the tree is an object 16 bytes in size (two 4-byte references and 8-byte data). Initially, a complete binary tree of 32,767 nodes was instantiated in the *caller* address space. Then a reference to the root object of the tree is passed to a remote method on a *remote* address space. The remote method traverses the tree in a *depth-first* manner, following the references. We measured the average time required to process one remote method call that retrieves the tree, varying the number of the nodes visited in the remote method. In this experiment we compared three cases, changing the closure size parameter in each case.

- **Case (A)** Closure size parameter is set to zero. The objects migrate to the callee on demand.

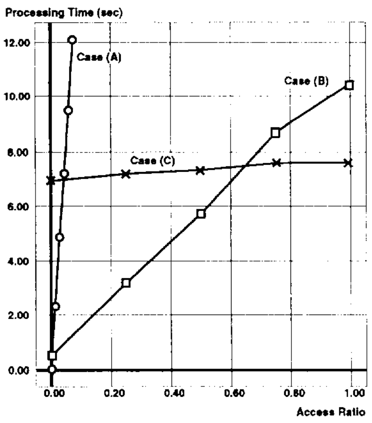


Fig. 8. Relationship between the Closure Size and Processing Time

- **Case (B)** Closure size parameter is set to 8 Kbytes. When an attempt is made to access an absent object, the 8 Kbytes transitive closure of the accessed object migrates to the callee. This transitive closure is retrieved in a *breadth-first* manner in the caller.
- **Case (C)** Closure size parameter is set to infinity. The entire tree (524,272 bytes) migrates to the callee at once.

Figure 8 shows the experimental result. The X-axis shows the ratios of the number of visited nodes to the total number of the nodes. The Y-axis shows the average processing time.

In case (A), the processing time is obviously bad. Since each object is 16 bytes in size in this experiment, the granularity of object migration is too fine to utilize the network bandwidth. The increased number of network communications degrades the execution performance. In case (C), the processing time is almost constant because the entire tree migrates to the callee at once. Case (B) shows the best processing time of the three for access ratios between 0.0 and 0.6. The improved performance of case (B) over case (C) is obtained because a relatively small number of objects are replicated. When the access ratio is larger than 0.6, case (B) becomes worse because of the increased number of communications. By adjusting the closure size parameter properly, the application programmers might improve the execution performance of their applications. In this experiment, the programmers should set the parameter to 8 Kbytes or so when the access ratio is less than 0.6. When the access ratio is larger than 0.6, the programmers should set the parameter larger.

4.3 Update

Finally we examine the effect of the HOS and HLS parameters described in Section 3.2. The HOS and HLS parameters limit the size of the hot object sets

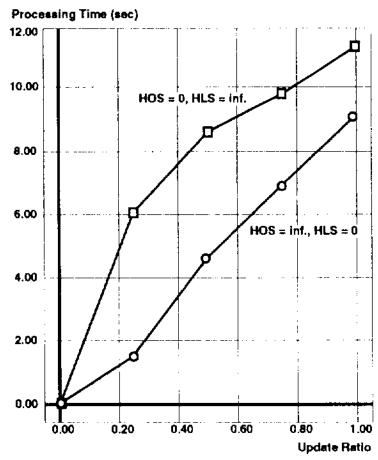


Fig. 9. Update Performance — In the case where locality of reference exists.

and the hot location sets, respectively. In this experiment, we use the same binary tree as in the previous section. To increase the effects of the HOS and HLS parameters, we set up the situation in which the entire tree is replicated on two address spaces A and B. In other words, each address space has its own replica of the identical tree. We can create this situation with the closure size parameter set to infinity, if the activity on the address space A passes a reference to the root object to the address space B; the entire tree migrates to the address space B with the migration of the activity.

We first examine the case where locality of reference exists. In this case, the activity on the address space A visits the nodes of the tree updating each visited node in the depth-first manner. Then the activity migrates to the address space B, traverses all the nodes of the replicated tree there, and returns to the address space A. We measured the processing time required from the activity migration to the address space B until it returns to the address space A, changing the ratios of the number of the updated nodes to the total number of the nodes. As shown in Fig. 9, the execution performance is better when the HOS parameter is set to infinity, since the objects modified in the address space A are accessed in the address space B. To examine the case where there is no locality, we carried out the same experiment as described above except that the activity does not traverse the tree at all on the address space B. The result is shown in Fig. 10. In this case, the execution performance is worse when the HOS parameter is set to infinity, since the hot object set transfers the objects that are not accessed in the address space B. The experimental results in this section validate our heuristics described in Section 3.2.

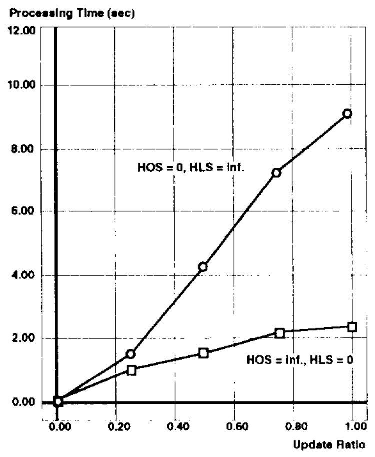


Fig. 10. Update Performance — In the case where there is no locality of reference.

5 Conclusion

We have described the design and implementation of the TRAP-DO distributed object system. TRAP-DO provides migratable distributed objects by applying the novel RPC technology integrated with virtual memory management. Compared with traditional distributed object systems, this integration brings about both the flexibility of proxy approaches and the instruction code level transparency of distributed shared memory (DSM) approaches. As a result, TRAP-DO naturally achieves object migration as DSM approaches, and is applicable to a heterogeneous environment as proxy approaches. The instruction code level transparency enhances the effect of object caching since it does not prevent compiler optimizations, and migrated objects are accessed without any additional overhead. The flexibility of proxies enables dynamic method binding in a distributed environment. To provide a natural and consistent view of distributed objects, TRAP-DO maintains the coherency of replicas and controls concurrency among multiple activities. The replica coherency protocol that provides one-copy semantics is simple and efficient because it makes use of the synchronous property of method invocations. The protocol for concurrency control takes an optimistic approach to avoid distributed deadlocks and additional network communications for distributed locks or semaphores. These protocols release the programmers from complex management of replicas.

Some interesting research issues still remain. The current protocol for concurrency control is a simple one. The primary drawbacks of the protocol is that it does not allow concurrency within a transactional session and that the programmers cannot make a decision what to do when a transactional session aborts. In principle, more elaborate protocols can be incorporated into TRAP-DO without sacrificing other features of TRAP-DO. We are currently designing a new protocol that allows a transactional session to create child transactions like nested transactions. In this new protocol, child sessions can execute concurrently, and the parent session can determine what to do when one of the child sessions aborts.

Another issue is to develop an optimal algorithm for taking the subset of the transitive closure of an object when the object migrates to a remote address space. If the access pattern of applications were precisely estimated, we could minimize the cost of network communication, but it requires predetermination of the access patterns of applications. One promising solution is to use hints provided by the programmers.

References

1. H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, Vol. 18, No. 3,, March 1992.
2. John K. Bennett. The design and implementation of distributed smalltalk. In *ACM OOPSLA'87*, pp. 318–330, 1987.
3. A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39–59, February 1984.
4. Vinny Cahill, Sean Baker, Chris Horn, and Gradimir Starovic. The Amadeus GRT - generic runtime support for distributed persistent programming. In *OOPSLA Proceedings 1993*, pp. 144–161. ACM, 1993.
5. Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *ACM OOPSLA'91*, pp. 1–15, 1991.
6. P. Dasgupta, R. LeBlanc Jr., M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*, Vol. 24, No. 11, pp. 34–44, Nov. 1991.
7. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conf. on Object-Oriented Programming (ECOOP)*, pp. 77–101, 1995.
8. Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software Practice and Experience*, Vol. 21, No. 8, pp. 757–785, August 1991.
9. J. Gosling and H. McGilton. The Java language environments: A White Paper. Technical report, Sun Microsystems, 1995.
10. Roger Hayes and Richard D. Schlichting. Faciliating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 12, pp. 1254–1264, December 1987.
11. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 109–133, February 1988.
12. 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, pp. pp. 119–143. Academic Press, 1993.
13. K. Kono, K. Kato, and T. Masuda. Transparent pointers in remote procedure calls. In preparation for submission.
14. 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*, pp. 142–151, 1994.

15. D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lan. The stanford dash multiprocessor. *IEEE Computer*, pp. 63–79, March 1992.
16. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321–359, November 1989.
17. A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *Proc. 20th ACM Symp. on Principles of Programming Languages*, pp. 99–112, January 1993.
18. H. Okamura and Y. Ishikawa. Object location control using meta-level programming. In *European Conf. on Object-Oriented Programing (ECOOP)*, pp. 299–319, 1994.
19. Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *Proc. IEEE Int. Conf. on Distributed Computing Systems*, pp. 198–204, 1986.
20. Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In *European Conf. on Object-Oriented Programming (ECOOP)*, pp. 191–204, 1989.
21. Sun Microsystems Inc. *External Data Representation Standard:Protocol Specification*, March 1990.
22. J. E. White. *Mobile Agents*. MIT Press, 1996. To appear.
23. S. Zhou, S. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 5, pp. 540–554, September 1992.