Distributed Shared Repository: A Unified Approach to Distribution and Persistency

Kazuhiko KATO† Atsunobu NARITA‡ Shigekazu INOHARA‡ Takashi MASUDA‡

† Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba, Ibaraki 305, Japan
email: kato@softlab.is.tsukuba.ac.jp

‡ Department of Information Science
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
email: {narita,ino,masuda}@is.s.u-tokyo.ac.jp

Abstract

This paper proposes an information management system providing distribution and persistency. By separating context *from virtual address space, our system has a unified approach for both distribution and persistency. The former is achieved by moving contexts between sites and the latter by moving contexts between virtual address space and persistent storage. Contexts include any information including data, program, and even the state of execution of a program. Contexts are stored persistently in a logical space termed the* distributed shared repository *(DSR for short). This paper proposes a programming model for DSR. Using the model, persistency, fine-grain mobility of information, and the passing of various distributed parameters can be obtained. The implementation and experimental performance of the system are also presented.*

1 Introduction

Both distributed and persistent processing are necessary in the construction of present information management systems. By distributed processing we mean several computational processes on different sites are able to interact with one another communicating and sharing information. By persistent processing we mean computation activity affecting the persistent store (typically a magnetic disk) that guarantees the storage of information regardless of whether the system is up or not. In fact, almost all business applications and computer-aided design applications require persistent information management, and recently such information management has been required to be adaptable to distributed environments. This is especially the case, in the growing areas of groupware or CSCW, where distributed persistent information management is indispensable.

This paper proposes a distributed and persistent information management system and describes its programming model and implementation. Through previous work on distributed systems [13, 4] and persistent systems [14, 12], we obtained an understanding that distributed processing and persistent processing have a crucial aspect in common: *the essence of both types of processing is information traffic between the address space and the outside.* In distributed processing, information is interchanged between different address spaces, while in persistent processing, information is transmitted between a volatile address space and a persistent store. This observation led us to seek a unified approach for the management of distributed and persistent information.

Prior to designing our system, we adopted a policy that the scheme would be independent of specific programming languages or special hardware. This policy is unique since most other attempts to design distributed persistent systems assume the use of specific programming languages or special hardware. Due to this, our system could easily be ported to many platforms and could be used by many programmers using arbitrary programming languages.

To invest information manipulated by application programs with both distribution and persistency, assuming no any language-oriented constructs, any state of computation at any time needs to be "frozen" and handled as data. For this reason, our system treats program execution *context* as a first class object. To ensure uniformity, all programs and data, persistent and passed on a distributed environment, are regarded as contexts. A context may include a program, data manipulated by the program, run-time information on stack(s), and the CPU program counter and registers. Contexts are stored persistently in a *distributed shared repository* (DSR for short). Although component of

0-8186-3770-6/93 $3.00

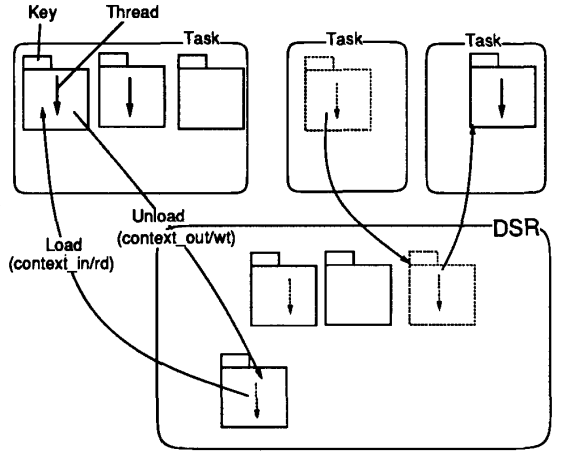

Figure 1: DSR Programming Model.

DSR is distributed to several sites, users simply see this as one persistent space. Each context in DSR has a unique and location-independent key, and the system is able to determine the context from any given key and to load the context to virtual address space of the user. When a context includes pointers, the system automatically revises pointers for the space concerned.

The remainder of this paper is organized as follows. Section 2 describes a programming model for DSR. Section 3 presents the manner in which distributed and persistent information management is performed using the model. Section 4 presents implementation of the model and Section 5 discusses the experimental results of implementation. Section 6 discusses related work. Finally Section 7 concludes this paper and suggests future work to be done.

2 DSR Programming Model

This section describes a programming model using DSR that provides persistency for arbitrary application programs in distributed computing environments.

2.1 Basic Concepts

Let us begin by introducing three basic terms: *task*, *context*, and *thread*. See Fig. 1. A *task* represents a virtual address space which is assumed to be addressed linearly. A *context* represents a state of computational activity. Also, at the same time, a context is a unit of memory loaded onto a task and unloaded from a task in the programming model. A *thread* represents the logical path of computational activity by a CPU, and a thread runs on the contexts loaded on a task. We assume that any number of threads can exist simultaneously in a task, and multi-threads on the task can proceed either in parallel or in pseudo-parallel. The notions of task and thread have been common in recent research operating systems, whereas the concept of context is novel and at the heart of the programming model.

Contexts are classified into three types according to the number of memory segments in them. The segments are *data*, *text*, and *CPU-state*. A data segment is a writable memory that can be manipulated by programs in a text segment. A text segment is a memory object that can be directly interpreted by CPU. The modification of this text segment is usually prohibited. A CPU-state segment is a writable memory object that stores a CPU-state which typically includes a stack to maintain the history of procedure calls, the current values of the program counter and registers of a CPU.

Although within a context 2^3 combinations of segments are possible, three combinations are usually significant. The three combinations are called Type I, II, and III contexts depending on the number of segments in them. A Type I context only includes a data segment representing data that does not include any machine-executable texts. Type I contexts are used to represent human-readable text files and binary data such as graphic images. A Type II context includes a text segment as well as a data segment. Type II contexts are used to represent abstract data types that encapsulates both internal data and operations, or program libraries generated by language compilers. A Type III context includes a CPU-state segment in addition to a data segment and a text segment. A Type III context is used to represent both an executable load module (that may have a null-initialized stack) and a program execution image. A Type III context is considered to be an active context since it keeps track of a thread, and vice versa, Type I and Type II contexts are passive contexts.

2.2 Basic Primitives

To manipulate contexts, a thread can issue four basic primitives: context_in, context_rd, context_out, and context_wt (see Fig.1). Every context is either loaded onto a task or stored persistently in DSR. the context_in and context_rd load a context from DSR onto a task, and context_out and context_wt unloads a context from a task to DSR. The primitives are named after Gelernter's TS primitives, and are designed on TS analogy, even though they are not for interprocess communication but for transmitting contexts between

a task and DSR. The signature of the primitives are:

context_in:	key → c_spec
context_rd:	key → c_spec
context_out:	(key, c_spec) → unit
context_wt:	(key, c_spec) → unit.

where unit is a trivial type whose only elements is void, and c_spec is a specification of a loaded context including the starting address and the size of the context. The context_in(key) retrieves DSR by specifying a key key. If a context having a key that matches key is found, the context is loaded onto the task of the thread that issues the primitive and the context is removed from DSR. At the time of loading, necessary adjustment to the content of the context on the task, including relocation of address reference, is performed. The details of the adjustment will be described in Section 4. If a matching context is not found, the thread issuing the primitive is blocked until the matching context is found. The context_rd(key) is similar to context_in except that context is *not* removed from DSR after the context is loaded onto the task.

The context_out(key, c_spec) unloads a context specified by c_spec and stores the context to DSR. The key specified by key is given to the context. The key must be unique in DSR. A thread can specify a predetermined special value, SELF, as an argument c_spec to unload the context on that the thread is running. The value SELF is used under two conditions. One is because it is a convenient way for a thread to specify the unloading of the context on that the thread is running. The other is to deal with the issuing of the context_out primitive in the middle of processing inter-context procedure calls. We have explained this using the example shown in Fig. 2. Assume that in a task there are only Type III Context A and Type II contexts B and C. When a thread on Type III Context A calls a procedure on Type II Context B, the thread on the procedure also calls a procedure on Type II Context C. Finally the thread on the procedure issues context_out(SELF, key), then a combination of Context A, B, and C is unloaded to DSR. We have termed this "domino" unloading. This combination is treated as a Type III context, and the key specified by key is given to the context in DSR. Using context_out the task is removed from the task after the context is unloaded to DSR. The context_wt is similar to context_out except that the context is *not* removed from the task and remains in the task even after unloading.

Besides the four basic primitives to manipulate contexts, we have assumed that a thread can communicate with other threads, which may resides on the same task or on another task at another site, using some method of communication such as message passing or remote procedure calls. (The programming model is independent of any specific communication method, and any communication method can be combined with it.) We have assumed that a linker can initially generate Type II and Type III contexts with initializing segments in them.

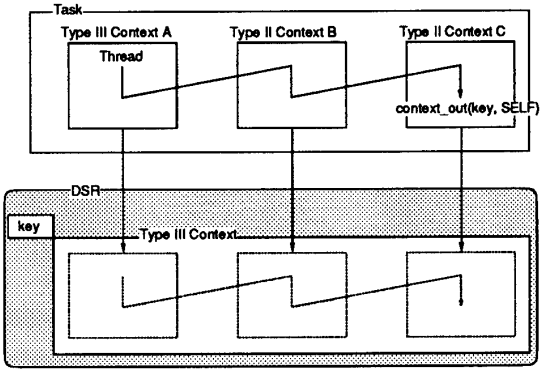

Figure 2: "Domino" unloading.

3 Utilization

Here, we demonstrate how the DSR programming model is used as a basis for various types of information management for persistency and distribution.

3.1 Adding Persistency to Programming Languages

Adding persistency to programming languages is an important research theme since it enables to model and manipulate persistent objects as well as objects on volatile main memory (see [2] for a survey of this field). The DSR programming model can be used as a basis to provide persistency for arbitrary programming languages.

See Fig. 3. A thread runs on the user program of a Type III context loaded to a task, and the thread is required to access data or procedures from a persistent object represented as the Type II context stored in a persistent store, i.e., DSR. Persistent object access can then be performed in the following steps (the numbers correspond to those in the figure):

(1) The thread issues context_in. Using this, the Type II context is loaded to the task.

(2) The thread can read or write data, or call procedures on the loaded persistent object.

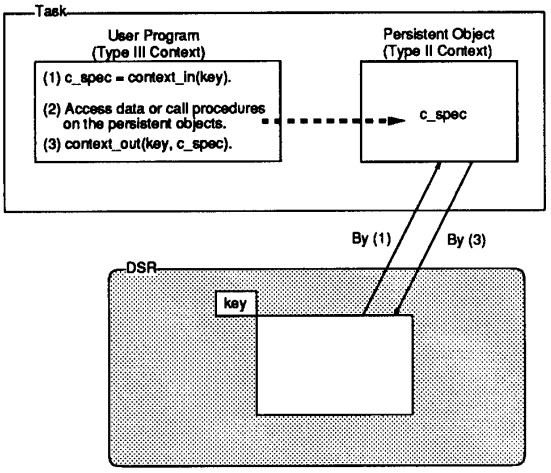

Figure 3: Persistent object access.

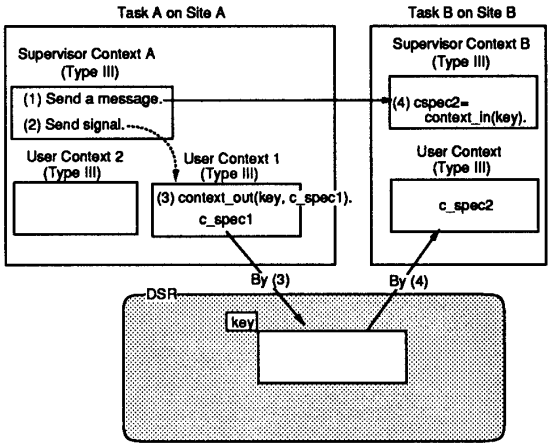

Figure 4: Context migration.

(3) After the required accesses have been attained, the thread issues context_out. After this the Type II context, which may be modified by the thread, is written back to DSR.

This example features two attractive aspects of the DSR programming model. First, any information on data and programs (on the Type II context), regardless of data type, can be persistent. The context_in and context_out operations may be regarded as "transaction begin" and "transaction commit" operations in usual database or persistent object processing. Furthermore, it is not difficult to develop a language processor (or preprocessor) that automatically inserts primitives without the user program explicitly issuing them. Such a method is described in [14, 18]. Second, although contexts stored in DSR may be physically distributed, they are transparent to user programs. The user program simply specifies a key that is independent of the physical location.

3.2 Context Migration

Process migration has attracted much attention from distributed system researchers. As a process has not been divided into a task, threads, and contexts in the prior distributed operating systems, entire address space has had to be moved from site to site. In the DSR programming model, because of division, *context migration* between existing tasks on different sites is possible. We can expect that context migration is faster than process migration due to the following two reasons. One reason is that, with context migration, the expensive operation of creating a new virtual address space is not required since the existing virtual address space (i.e. task) on the target site can be used. The other reason is that, with context migration, important fragments of data or procedures to be migrated can be cut from the virtual address space, and only the fragment required (i.e. a context) is migrated to the target site. Although such advantages are obtained through "fine-grained mobility" in the Emerald system [10], context migration is more generic in the sense that our scheme does not assume either specific programming languages or object-oriented paradigms. Performance gains using our current implementation of context migration will be discussed in Section 5.2.

We will now describe how context migration is accomplished transparently to migrated contexts. See Fig. 4. For the sake of simplicity, we have assumed that only one task, Task A, is created on Site A and two Type III user contexts 1 and 2 exist. We have also assumed that user context 1 is selected to be migrated to Task B on Site B. A Type III supervisor context exists on Task A. Context migration is then performed using the following four steps:

(1) The thread on supervisor context A send a message to supervisor context B so that a context with key will be migrated to Task B.

(2) The thread on supervisor context A sends a signal (or software interrupt) to user context 1. The signal handler defined in user context 1 determine the state of context for migration*.

*In the DSR programming model, all information closed within the context is guaranteed to persist and is migrated via a

(3) After this, the signal handler issues context_out unloading the context itself from Task A to DSR.

(4) The thread on supervisor context B issues context_in after receiving the key in Step (1). Just after both context_out in Step (3) and context_in in Step (4) are issued, user context is loaded from DSR to Task B.

It would be possible that operating system kernels, instead of supervisor contexts, deal with context migration. Context migration controlled by kernels needs modification to the kernels, while migration using supervisor contexts does not require any kernel modification.

3.3 Distributed Parameter Passing

In designing any distributed programming system such as object-based systems or RPC (remote procedure calls)-based systems, the parameter passing method is one of the most important concerns. With the DSR model distributed programming systems can take advantage of passing contexts as parameters. Here, we will explain context passing between threads communicating with RPC. In the literature, the distributed parameter passing method explained below is termed *call-by-visit* [10].

See Fig. 5. We have assumed that the caller context is on Task A in Site A, and that the callee context is on Task B in Site B. A Type II context passed as an argument in RPC is initially assumed to be on Task A.

(1) The thread on caller context issues context_out to unload the argument context from Task A to DSR.

(2) The thread on the caller context marshals the arguments including the key and issues a remote procedure call to the callee context.

(3) After the thread on the callee context receives the call, it un-marshals the arguments.

(4) The thread on the callee context issues context to load the argument context.

(5) After the processing of the body of the called remote procedure on the callee context, the thread on the callee context replies to the remote procedure call with marshaled return values.

network. However, like process migration schemes, some state relating to an operating system kernel or other contexts such as logical communication links need to be settled depending on the underlining system on which the DSR programming model is built. Detailed discussion about the settlement in process migration is found in [7].

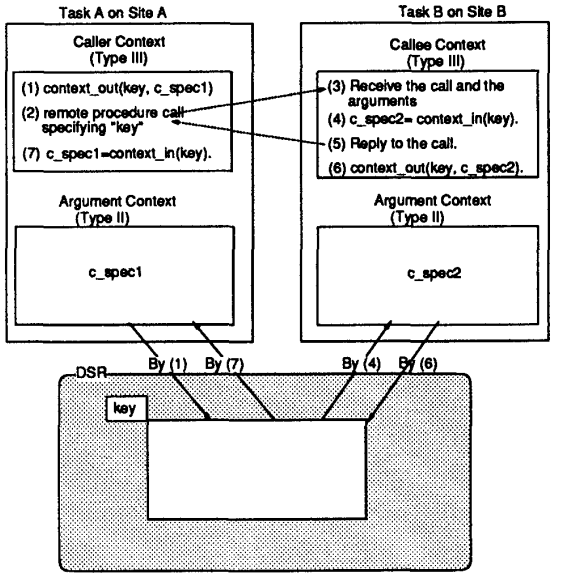

Figure 5: Call-by-visit.

(6) To return the argument context, which might be modified, to Task A, the thread on the callee context issues context.

(7) By issuing context_in, the thread on the caller context receives the argument context.

4 Implementation

This section describes a way to implement the DSR programming model based on our current implementation. The implementation does not assume the use of special hardware.

There are three crucial aspects at the heart of the implementation. The first is the need for technique for managing multithreads in a task. The second is general context relocation without assuming any language-level constructs. The third is the name resolution for a DSR. Recently, a few researchers have developed efficient multithread management techniques (see [1, 9, 16] for example). Since this issue is well understood now and is independent of both the context relocation technique and the construction of a DSR, we concentrate on the second and third issues in the rest of this section.

Although implementation has achieved in the course of developing the XERO experimental distributed operating system [11, 9], it does not depend on the special features of XERO. One outstanding feature in its structure distinguishes XERO from other

operating systems; every task has a program module in its user address space, which supervises the execution of all programs in the task. This program module is called a *task supervisor* (TSV). When a task is created, the TSV is invoked, and a thread of control of the CPU is passed to it first.

4.1 Relocating Contexts for Loading and Unloading

As described in Section 2, multiple contexts, which may include text segments (i.e., compiled binary codes), can be dynamically loaded to a task and unloaded from a task. Although many relocation techniques were proposed previously, a novel relocation technique is required since, in DSR, contexts are loaded and unloaded dynamically and repeatedly from and to the outside of a virtual address space.

Dynamic loading and unloading facilities for contexts require the relocation of contexts so that loaded contexts will function for any task. Since Type I and II contexts can be regarded as a subset of Type III contexts, we will describe a way to relocate Type III contexts.

To attain machine-independent relocation of Type III contexts, we must achieve relocation of text, data and CPU-state segments (including stacks), and resolution of intercontext pointer references. In the following we describe these issues in the order.

(a) Relocation of text segments

Relocatable text segments are attained simply by employing the CPU *register-relative* addressing mode to access all named memory cells (i.e., symbols in programming languages). Let us examine modifications to the GNU C compiler [21] on a SONY NEWS workstation, which has a M68030 CPU. Our modifications are the following:

1. The `a5` register of the M68030 CPU is used as a "base" register, in which the starting address of a loaded context is set.

2. All references to the named memory cells are performed using the register-relative addressing mode of the CPU.

(b) Relocation of data segments

A data segment is internally composed of two areas: *static data area* and *heap area*. The all things to do to relocate a data segment is to find all the occurrences of pointers and to adjust them to the new location in a task. To find the occurrences of pointers in a static data area, our implementation uses a symbol table generated by a compiler. When the TSV loads a context to the task, the difference is added between the starting address of the currently existing task and the previously existing task, to every pointer occurrences.

The occurrences of pointers in a heap area, on the other hand, cannot be determined in advance and may change dynamically. Therefore, either of language processor or programmers must provide explicit typing information for the heap area. In the case of strongly statically typed languages such as Pascal or Standard ML, the compilers can produce such information from type declaration or type inference. In the case of weakly statically typed languages such as C, the programmers must provide dynamic typing information. In our implementation using GNU C compiler, a TSV primitive `typing` is used for this purpose.

```
typing(ptr, type, elements)
char *ptr;     /* pointer to the variable */
char *type;    /* type of an element */
int elements; /* number of elements */
```

This specifies that the heap area starting from the `ptr` address contains `type` related data, and the data number is designated by `elements`.

(c) Relocation of CPU-state segments

Here we have assumed that a CPU-state consists of a *stack*, a *program counter* and *registers* of CPU. All information on a stack can be relocated by analyzing the stack frame and symbol table. Interestingly, the relocation of a program counter and registers is attained by the relocation of a stack. By saving them onto stacks at the unloading time, they are relocated with the stack relocation mechanism mentioned above.

(d) Resolution of Intercontext Reference

After loading and relocating a context in a task, it may be required to resolve reference to data or procedures in other contexts. Logically, every address in a context can be specified by a key specifying the context uniquely and the offset from the top of the context. Since a CPU can access addresses only within a task directly, a logical "long format" address must be mapped and translated to an accessible "short format" address (a 32-bit address in the M68030 CPU); this has become known as *pointer swizzling* (see [3] for more explanation of this). If we assumed that DSR was used with a single language, we could implement an automatic pointer swizzling, with which a procedure could transparently reference data and procedures of other contexts as well as of its context. Our

intention, however, is to make a language-independent DSR. Our current support for intercontext reference is rather basic and primitive; data or procedures can be logically referenced from other contexts provided a symbolic name is associated and the symbol name is registered in the symbol table of the context. (Registration of symbol names in a symbol table is usually performed by language compilers.)

The address associated with a symbol name is retrieved by the TSV primitive **get_address(c_spec, symbol_name)**. Prior to issuing this primitive, the context containing the symbol specified by **symbol_name** must have been loaded. The argument **c_spec** specifies the context. The primitive returns the address of the symbol specified by **symbol_name** of the loaded context. When a programmer wants to call a procedure of other contexts, he can call the procedure with the TSV primitive **call_proc**. Here is an example usage of these primitives.

```
int ret;
char *procptr;
        :
c_spec = context_in(key);
procptr = get_address(c_spec, "proc");
        :
/* Intercontext procedure call */
ret = call_proc(c_spec, procptr, args...);
        :
```

The primitive **call_proc** properly sets the "base" register (the **a5** register of M68030 in our implementation) of a CPU, passes the specified arguments to the procedure specified by **procptr**, and transfers its thread to the context specified by **c_spec**. When the procedure execution is complete, the return value of the procedure is passed back to the calling (original) context in the same manner as when called.

Although these primitives are low-level, more high-level service such as automatic pointer swizzling could be implemented by a dynamic linker that joins symbol tables of loaded contexts with these primitives.

4.2 Name Resolution

Name (key) resolution function of DSR should satisfy the following requirements.

- Keys given to contexts should be independent of location so that sites storing contexts are reconfigurable when contexts are tidied up beyond site boundaries, when disk drives are moved between sites, or when some of sites are removed for repairs.
- From location independent keys, the current physical location of contexts should be found with low overheads.
- The implementation of DSR should be available in or ported to widespread distributed environments with a little effort.

After careful consideration of these required conditions, we decided to implement the name resolution function using the *prefix table* technique originally proposed by B. Welch and J. Ousterhout [22]. To system administrators, DSR is a collection of *c-domains* (shortened for context domains) , which are units of physical storage management. Physically, a c-domain is a set of disk storage blocks residing in one site, and a site may have several c-domains. A c-domain can be moved between sites, but the movement of a member context of a c-domain to another c-domain is inhibited without changing the key of the context.

To users, the boundary of c-domains is transparent, and a single logical name space is provided for the context keys. To make DSR available in current widespread distributed environments, our current implementation adopted a Unix-compatible name space for context keys.

5 Experiments

This section presents some experimental results based on our implementation of the DSR programming model using SONY NEWS workstations, each of which has a MC68030 CPU and a MC68881 FPU with 8 MB main memories. We measured the actual performance of context loading and unloading, and context migration between sites. For the measurement, we used a version of the DSR system running on the 4.3BSD Unix.

5.1 Performance of Context Loading and Unloading

To evaluate the performance of loading and unloading contexts, we repeated the loading and unloading of a Type III context to and from a task one hundred times, varying the size of the context (no effective work was done by the contexts). The examined contexts were stored at the same site where the task existed and no network communication occurred during this experiment. The source codes for the examined contexts were extracted from 4.3 BSD UNIX utility programs written in C language. The C programs were compiled by a GNU C compiler modified as described in Section 4.

Table 1 shows the average costs in terms of time for one loading or unloading operation. In the table, user time is the time required for relocation, and system time is the cost of executing disk file I/O. The response time in the table is the total time in terms of user time,

Table 1: Performance measurement of loading and unloading contexts (time in milliseconds).

Program	Context Size (KB)	Size of Symbol Table (KB)	No. of Pointers	User Time	System Time	Response Time
(a)	16	1	2	2.2	21.7	151.0
(b)	37	6	92	35.2	37.6	220.7
(c)	64	11	181	93.0	59.9	353.5
(d)	125	19	1,406	197.0	98.2	692.9

Table 2: Performance measurement of context migration.

Program	Context Size (KB)	Size of Symbol Table (KB)	No. of Pointers	Response Time without task creation (milliseconds)	Response Time with task creation (milliseconds)
(a)	16	1	2	20.01	20.76
(b)	37	6	92	34.63	46.10
(c)	62	9	43	67.95	107.30
(d)	125	19	1,406	123.45	189.18
(e)	316	53	2,326	328.00	450.50
(f)	407	62	959	428.20	698.50
(g)	625	110	912	707.50	1,725.30

system time, and physical disk operation time to load and unload a context. All overheads to relocate a context are included in user time since all relocating operations are executed in the user mode. In any case, user time is less than 30 % of response time. Our current implementation does not use an efficient table management technique. Performance using a large symbol table and a large number of pointers can be improved by using a more efficient table management technique such as hashing. Since we expect that loading and unloading are generally not performed as frequently, we are able to state that the relocation mechanism can withstand practical use.

5.2 Performance for Context Migration

To examine how the separation of contexts from a virtual address space affect performance in a distributed environment, we measured the performance of context migration between two sites, A and B, connected via Ethernet. As a network protocol TCP/IP was used. Contexts are migrated from Site A to Site B and from Site B to Site A in succession. During each migration, context relocation is performed. Succession was repeated one hundred times and the average response time for one migration was calculated.

Table 2 shows the results. In the table "Time without task creation" designates the average response time to perform context migration between two fixed tasks, one of which is on Site A and the other is on Site B. In contrast, "Time with task creation" designates the average response time in performing context migration while creating a virtual address space at each migration. These two setting are to estimate the performance gain of context migration compared to process migration. The difference in response times between the former and the latter cases are greater when sizes of contexts are larger. This appears to be because the virtual memory management of 4.3BSD Unix requires more time for a larger virtual address space. From the results, we can estimate that context migration is attractive for performance in the larger contexts.

6 Related Work

This work relates to segmentation, object-oriented, and distributed shared memory.

6.1 Segmentation

Historically, the first attempt to integrate information management on both virtual address spaces and persistent storage was the *segmentation* mechanism [6] originally developed for the Multics operating system, although the system is not concerned

with distributed processing at all. Our DSR approach might be regarded "distributed segmentation," although DSR does not assume the use of any special hardware and can be ported on almost all CPU architectures even those not having segmentation hardware.

6.2 Object-Oriented Systems

Recently, several researchers have been vigorously studying the building of distributed systems based on object-oriented concepts. Clouds [5], Emerald [10], SOS [19], COOL [8], Apertos [23] and Galaxy [20] are examples. Some of these provide object mobility apart from virtual address spaces, and some can create persistent objects. The most distinctive feature of the DSR approach compared to the object-oriented approach is language independence. Most of the object-oriented approaches require a prerequisite to use either extensions of existing object-oriented programming languages or novel languages designed for their own systems. Our DSR approach does not requires the use of such language dependencies. Any information represented with any programming language can be treated as contexts, with distribution and persistency attained to them. Certainly, it is possible to use DSR as a basis of building object-oriented distributed and persistent systems.

ObjectStore, an object-oriented database system, uses the memory-mapped file technique to make any portion of a virtual address space persistent. In [15], the developers reported that such a system requires a pointer relocation technique and outlined their technique, which is similar to the one described in Section 4. Their technique is limited to a particular object-oriented language (C++) and never relocates pointers in stacks, unlike our location technique. Hence, ObjectStore cannot deal with active objects within which threads of control exist, unlike our system.

6.3 Distributed Shared Memory

Using our scheme, information can be shared between the virtual address spaces on several sites. Such information sharing relates to the *distributed shared memory* (DSM for short) scheme, which has recently caught much attention from distributed system researchers (see [17] for recent surveys). DSM is useful since it enables access to virtual memory over machine boundaries. DSM is also useful because page accesses are transparent in application programs. Although DSM can be considered as a basic technique for future distributed systems, it is unable to provide information management such as logical naming and access control. However, the scheme described in this paper does enable such information management.

The relationship between DSR and DSM is almost as close as the relationship between the *paging* and *segmentation* of virtual memory management. Paging is a technique to extend addressable memory to secondary storage space, and is transparent to both application and programmers. Segmentation is used to manage and give persistency to information. In a sense DSM is a distributed extension of the paging technique, while DSR is a distributed extension of the segmentation technique. A context corresponds to a segment and DSR corresponds to a file system storing segments.

7 Conclusion

This paper makes two important claims. First, the information unit termed context can be separated from the virtual address space without assuming language-level constructs, and the unit can be transferred between virtual address spaces on different sites. Second, the unit, or context, is stored in a logical space, namely DSR and a context can be passed between virtual address spaces through DSR. Application programs can access a context by a location-independent logical key regardless of the physical location of the programs. With these two mechanisms, application programs can gain access to any distributed and/or persistent information.

The scheme described in this paper should be extended according to the following ways. First, the virtual memory technique should be incorporated in the DSR to rectify a shortcoming of our current implementation, in which a whole context is physically copied between a task and the DSR. With a virtual memory technique, only referenced pages are copied from the DSR to a task, and only modified pages are written back to the DSR. We think this is an efficient way to combine the distributed shared memory techniques with the DSR. Second, sophisticated access control and concurrency control of contexts should be studied. Considering incorporation of the capability and transaction concepts could be good starting points. Finally, reliability and efficiency should be improved by introducing replication and caching techniques.

Acknowledgment

The authors thank Shigeru Chiba for valuable discussions for an earlier draft of this paper. Comments by the anonymous referees were very useful to clarify the description of this paper.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proc. of 13th ACM Symp. on Operating Systems Principles*, pp. 95–109, October 1991.

[2] M. P. Atkinson and P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, Vol. 19, No. 2, Jun 1987.

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

[4] S. Chiba, K. Kato, and T. Masuda. Exploiting a weak consistency to implement distributed tuple space. In *Proc. IEEE 12th Int. Conf. on Distributed Computing Systems*, pp. 416–423, Jun. 1992.

[5] 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.

[6] J. B. Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM*, Vol. 12, No. 4, pp. 589–602, Oct. 1965.

[7] F. Douglis and J. Ousterhout. Transparent process migration: design alternatives and the Sprite implementation. *Software Practice and Experience*, Vol. 21, No. 8, pp. 757–785, Aug. 1991.

[8] S. Habert and L. Mosseri. COOL: kernel support for object-oriented environments. In *Joint ECOOP/OOPSLA Conf.*, Ottawa, Oct. 1990.

[9] S. Inohara, K. Kato, A. Narita, and T. Masuda. A thread facility based on user/kernel cooperation in the XERO operating system. In *Proc. IEEE 15th Int. Computer Software and Applications Conference*, pp. 398–405, Tokyo, Sep. 1991.

[10] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Trans. Computer Systems*, Vol. 6, No. 1, pp. 109–133, Feb. 1988.

[11] K. Kato, S. Inohara, A. Narita, S. Chiba, and T. Masuda. Design of the XERO open distributed operating system. *Journal of Information Processing*, Vol. 14, No. 4, pp. 384–397, Apr. 1991. Special issue on new operating systems.

[12] K. Kato and T. Masuda. Persistent caching: An implementation technique for complex objects with object identity. *IEEE Trans. Software Engineering*, Vol. 18, No. 7, pp. 631–645, Jul. 1992.

[13] K. Kato, T. Masuda, and Y. Kiyoki. A comprehension-based database language and its distributed execution. In *Proc. IEEE 10th Int. Conf. on Distributed Computing Systems*, pp. 442–449, Paris, May 1990.

[14] K. Kato and A. Ohori. An approach to multilanguage persistent type system. In *Proc. IEEE 25th Hawaii Int. Conf. on System Sciences*, Vol. II, pp. 810–819, Jan. 1992.

[15] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Comm. of the ACM*, Vol. 34, No. 10, pp. 50–63, Oct. 1991.

[16] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proc. of 13th ACM Symp. on Operating Systems Principles*, pp. 110–21, October 1991.

[17] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *IEEE Computer*, Vol. 24, No. 8, pp. 52–60, Aug. 1991.

[18] 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, Jan. 1993.

[19] M. Shapiro and P. Gautron. Persistence and migration for C++ objects. In *European Conf. on Object-Oriented Programming (ECOOP)*, Jul. 1989.

[20] P. K. Sinha, M. Maekawa, K. Shimizu, X. Jia, H. Ashihara, N. Utsunomiya, K. S. Park, and H. Nakano. The Galaxy distributed operating system. *IEEE Computer*, Vol. 24, No. 8, pp. 34–41, 1991.

[21] R. M. Stallman. *Using and porting GNU CC.* Free Software Foundation, Inc., 1991.

[22] B. Welch and J. Ousterhout. Prefix tables: a simple mechanism for locating files in a distributed system. In *Proc. IEEE Int. Conf. on Distributed Computing Systems*, pp. 184–189, 1986.

[23] Y. Yokote. The Apertos reflective operating system: the concept and its implementation. In *Proc. ACM OOPSLA'92*, pp. 414–434, Oct. 1992.