Distributed Operating Systems Based on a Protected Global Virtual Address Space

John B. Carter, Alan L. Cox, David B. Johnson, and Willy Zwaenepoel

Rice University
Department of Computer Science
P.O. Box 1892
Houston, TX 77251

Abstract

With the advent of the 64-bit microprocessor, the virtual address space supported by a workstation will be large enough to permit the use of a single shared address space spanning a network of workstations as the primary abstraction provided by a distributed operating system. In such a system, built upon a software distributed shared memory, the programmer has considerable flexibility when choosing a mechanism for interprocess communication. This flexibility permits the programmer to make a case-by-case choice between simplicity and performance when both goals are not simultaneously achievable. With the inclusion of mechanisms supporting *protection* and *fault tolerance*, we believe that such a system can provide the advantages of conventional message-based distributed operating systems (e.g., multiple protection domains, hidden data abstractions, simple client-server interface, and failure isolation), in addition to several other benefits (e.g., easy sharing of complex data structures between processes, transparent replication of server functions, and a uniform interface for all communication).

1 Introduction

Many researchers have developed distributed shared memory (DSM) systems that enable a *single* shared-memory parallel program to execute on multiple machines [2, 3, 4, 9, 11]. The typical motivation behind DSM is to reduce the effort required to program distributed memory multicomputers. However, no existing DSM provides the protection necessary for multiple, potentially mistrusting programs to safely coexist within the same address space.

Other research projects have explored the use of a single address space as a shared name space. In these systems, memory access serves as the primary mechanism for interprocess communication [1, 10, 12]. Most of these systems are intended to execute on a single machine.

We believe an efficient implementation of a single address space across *multiple* machines is possible by combining high-performance DSM and RPC mechanisms to provide both data movement and function shipping. This belief is based on the following observations. First, as processor speeds increase relative to network speeds, the ability to move or replicate shared data takes on increased importance. However, the provision of function shipping is necessary when the shared data should not be moved or replicated. Second, implementations of relaxed memory consistency models, such as *release consistency* [6], that are specifically designed for use by software DSM can significantly reduce the communication that occurs between machines sharing data [8]. Munin [3] has demonstrated that a DSM program can achieve performance comparable to its message passing equivalent. Third, with the advent of 64-bit microprocessors, the address space is large enough to name all potential objects in a distributed system.

All programs in our proposed system execute in the same global address space, so a particular virtual address points to the same object for all running programs. Like a DSM, the operating system would be responsible for keeping shared objects consistent. Most current operating systems combine the notion of a *protection domain*, a set of data and the operations that can be used to access it, and an *address space*, the mapping of addresses to objects. Thus, because independent programs execute in independent address

0-8186-2555-4/92 $03.00 © 1992 IEEE

spaces, they are protected from malicious or faulty modification of their private data. However, a protection domain in our proposed system corresponds to a set of address ranges in the global address space. Each thread in the system maintains its own view of the global address space, using conventional virtual memory hardware to map portions of the address space into the thread's protection domain. Accesses to protected portions of the global address space would force the accessing thread to change protection domains to the domain associated with the data. Inherent in such a design is our belief that the notions of an address space and a protection domain should be decoupled in future operating systems.

Supporting a protected global address space in the operating system provides several potential benefits. When the client and the server execute within independent address spaces, neither the client nor the server can pass or return pointers to objects within their address space. Thus, all of the parameters (results) must be translated into a message and recreated at the receiving end, a potentially expensive operation when the parameters (results) involve complex data structures involving arbitrary pointer chains such as trees, DAGs, or hash tables. Since virtual addresses have the same meaning in all running programs, complex data structures can be shared in our proposed system, eliminating the overhead involved in marshaling the parameters (results). Furthermore, because pointers have the same meaning on both sides of the procedure invocation, the semantics would be identical to those of conventional procedure calls, whereas an RPC has a slightly different semantic meaning than a conventional procedure call.

Another benefit of our proposed design is that by decoupling the notion of an address space from that of a protection domain, we provide the capability to *transparently* replicate server functions on the client machines. In a conventional distributed operating system, the only threads that can access the private data of a server are the actual server threads. Thus, any client that wanted to perform some operation on this private data would be required to send a message to the server and wait for a server thread to perform the operation and send a response. In our model, a client machine may execute the server code directly or perform an RPC to the machine that contains the data. This mechanism is discussed in more detail in Section 2. If the server code does not perform an RPC, then the global address space and associated consistency mechanisms allow the client thread to directly execute the operation, which might involve *data shipping* to load the portions of the server data that are accessed and do not already reside on the local machine. The ability to perform both function shipping and data shipping allows our proposed system to potentially achieve improved performance over conventional systems that provide only function shipping.

2 Protected Global Address Spaces

Most single address space systems have relied on the programming language and compiler to provide safety. However, we do not wish to restrict the programmer's choice of languages, and we therefore take a different approach. Decoupling the notions of a protection domain and an address space is fundamental to our proposal. The existence of a shared global address space does *not* mean that all parts of the global address space can be accessed by any thread in the system. We envision allowing portions of the address space to be protected so that access to these portions of the address space must occur via trusted routines that access this data in a controlled fashion. An outline of how this might be implemented follows.

Our proposed system supports *trusted* and *untrusted* clients, *trusted* and *untrusted* servers, and *data shipping* and *function shipping* invocations. A trusted client or server can directly access anything in the other's protection domain. A trusted client calling a trusted server incurs no overhead beyond a trivial stub executing a single branch instruction. An untrusted client (server) only has access to the parameters (results) and must perform a trap to change protection domains when the client-server protection boundary is crossed. The stubs for a function shipping invocation are analogous to RPC stubs. To reduce the number of messages passed on a function shipping invocation, if the server specifies that parameters are passed by reference, the referenced pages can be bundled with the call.

2.1 Binding and Invocation

When a program is loaded into the global address space, references to unresolved services are linked in. This linking is performed by sending a request to the global name server with enough information for the name server to determine the desired services, the identity of the invoking thread, and the type of machine on which the thread is executing (for heterogeneity). After authenticating the request, the name server maps

service stubs (and potentially a code segment) into the program's protection domain. The name server will include linking commands that will allow all unresolved external references within this segment to be resolved once at bind time, after which the code segment can be re-used very cheaply. However, function calls that occur through pointers will cause page faults and run-time linking.

At run-time, the vehicle for invocation is determined by the stub mapped by the name server into the client's protection domain. This could be implemented by laying out the server code segment as specified in Figure 1. A pair of *stub segments* come at the beginning of each server segment. The first contains the stubs for operations that are performed remotely via function shipping, and the second is for operations that are performed locally via data shipping. Following the stubs is the segment that contains the actual code used to execute the operations. The server specifies which operations involve data shipping and which involve function shipping when it registers itself with the name server. This is appropriate because the server has the most semantic information about how best to efficiently access its data structures.

In the example at the top of Figure 1, the server code and data segment are inaccessible from the untrusted client's protection domain, so the stub code must execute a trap to change protection domains to the server's before jumping to the server function. The example at the bottom of Figure 1 shows that a trusted client's protection domain incorporates the server's protection domain so that the trusted client can access the server data structures directly. Trusted and untrusted servers are handled analogously, which gives the client control over whether or not the server can follow pointers within the client's address space.

2.2 Server Specification

When a program wishes to export a set of services, it registers these services with the global name server. This process is analogous to the way in which servers in an RPC system register themselves and specify the services that they support. A server can specify the protection domains (users) that are permitted trusted access to its protection domain. The run-time linkable code segments and associated relocation commands can be automated as part of the post-compilation process. Heterogeneity can be supported by installing multiple executables with the name server.

2.3 Fault Tolerance

Despite the tighter coupling between processes implied by the single address space, we believe that the same degree of fault tolerance as available in conventional client-server systems can be provided at a similar cost. At the underlying system level, each machine in the distributed system has a separate memory and all communication between machines takes place by messages. These messages may be caused either by RPCs or by the protocols used to maintain memory consistency of the global address space. By integrating the fault-tolerance support with the RPC and consistency protocol support, existing distributed system fault-tolerance approaches [7, 5] can be used with minor modification. In particular, the existence of the shared

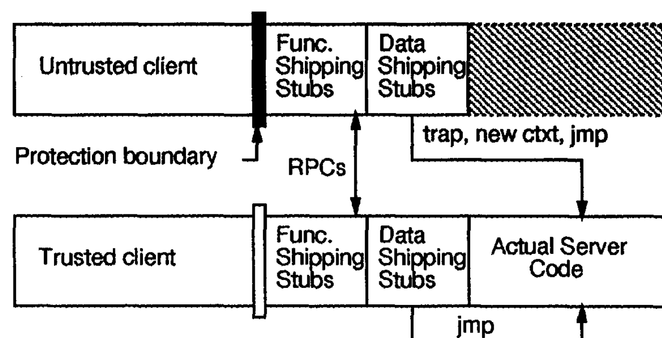

Figure 1 Layout of Server Stub and Code Segments

global address space does not force the creation of a large, system-wide checkpoint, since the underlying implementation uses physically separate memories and communication paths that can be recorded.

3 Related Work

A number of single address space systems have been built over the years [1, 10, 12]. Psyche is perhaps the most similar to our proposed system. However, differences exist in the cross-domain communication mechanism. Psyche uses page faults to trigger cross-domain calls rather than explicit traps by stubs. We believe that the use of stubs can reduce the cost of cross-domain communication. Another characteristic that distinguishes our system is its support for both data shipping and function shipping.

4 Conclusions

We believe an efficient implementation of a single address space across multiple machines is possible by combining high-performance DSM and RPC mechanisms to provide both data movement and function shipping. Such a system provides the capability to transparently replicate server functions on the client machines, and also supports call-by-reference semantics for RPCs. The choice of invocation style (data shipping versus function shipping, trusted versus untrusted) is under software control, which gives programmers the flexibility to use either RPC or shared memory semantics.

References

[1] R. Atkinson, A. Demers, C. Hauser, C. Jacobi, , P. Kessler, and M. Weiser. Experiences creating a portable Cedar. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989.

[2] H.E. Bal and A.S. Tanenbaum. Distributed programming with shared data. In *Proceedings of the 1988 International Conference on Computer Languages*, pages 82–91, October 1988.

[3] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[4] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[5] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers Special Issue On Fault-Tolerant Computing*, 41(5), May 1992. To appear.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

[7] D.B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Rice University, December 1989.

[8] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. To appear at the 19th Annual International Symposium on Computer Architecture, May 1992.

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

[10] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[11] U. Ramachandran and M.Y.A. Khalidi. An implementation of distributed shared memory. *Distributed and Multiprocessor Systems Workshop*, pages 21–38, 1989.

[12] M.L. Scott, T.J. LeBlanc, and B.D. Marsh. Design rational for Psyche, a general-purpose multiprocessor operating system. In *1988 International Conference on Parallel Processing*, pages 252–262, August 1988.