

Distributed Shared State

(position paper) *

Michael L. Scott,
DeQing Chen, Sandhya Dwarkadas, and Chunqiang Tang

Computer Science Department
University of Rochester
{scott,lukechen,sandhya,sarmor}@cs.rochester.edu

Abstract

Increasingly, Internet-level distributed systems are oriented as much toward information access as they are toward computation. From computer-supported collaborative work to peer-to-peer computing, e-commerce, and multi-player games—even web caching and Internet chat—applications devote a significant fraction of their code to maintaining shared state: information that has dynamic content but relatively static structure, and that is needed at multiple sites. We argue that tools to automatically manage shared state have the potential to dramatically simplify the construction of distributed applications and, in important cases, to improve their performance as well. We discuss the characteristics that such tools must possess, placing them in the context of past work on distributed file systems, distributed object systems, and software distributed shared memory. We present the InterWeave system as a prototype implementation, and discuss its strengths and limitations.

1. Introduction

Most Internet-level applications are distributed not for the sake of parallel speedup, but rather to access people, data, and devices in geographically disparate locations. Increasingly, these programs are oriented as much toward information access as they are toward computation. E-commerce applications make business information available regardless of location. Computer-supported collaborative work allows colleagues at multiple sites to share project

design and management data. Multi-player games maintain a distributed virtual environment. Peer-to-peer systems are largely devoted to indexing and lookup of a continually evolving distributed store. Even in the scientific community, so-called GRID computing [10] is as much about finding and accessing remote data repositories as it is about utilizing multiple computing platforms.

We believe that distributed systems will continue to evolve toward data-centric computing. We envision a future, for example, in which users enjoy continuous access to an “intuitive” digital assistant that manages all their data, models their intent, and suggests or carries out actions likely to satisfy that intent.¹ Like today’s PDAs, an intuitive assistant will have an interface that travels with the user, but unlike these devices it will continuously monitor its environment, offload expensive computations, and access information spread across the Internet. In effect, the PDA of the future will serve as the hub of a sophisticated distributed system with enormous amounts of shared state: sound and video recordings of the physical environment, records of past experience, and commonsense and task-specific knowledge bases.

Today’s systems employ a variety of mechanisms that might underlie distributed shared state. At one extreme are distributed file and database systems such as AFS [19], Lotus NotesTM, CVS [3], or OceanStore [16]. For the most part these are oriented toward external (byte-oriented) data representations, with a narrow, read-write interface, and structure imposed by convention. Data in these systems must generally be converted to and from an in-memory representation in order to be used in programs. At the other extreme, distributed object systems such as CORBA [18], .NETTM, PerDiS [9], Legion [11], and Globe [23] present data in a structured, high-level form, but require that pro-

*This work was supported in part by NSF grants CCR-9988361, CCR-0204344, CCR-0219848, ECS-0225413, and EIA-0080124; by equipment grants from Compaq, IBM, and Sun; and by the U.S. Department of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460.

¹For further discussion of this topic, see www.cs.rochester.edu/research/intuitive/.

grams employ an object-oriented programming style. While some of these systems do allow caching of objects at multiple sites, performance can be poor.

By contrast, the shared memory available within cache-coherent multiprocessors allows processes to share arbitrarily complex structured data safely and efficiently, with ordinary reads and writes. Many researchers (ourselves among them) have developed software distributed shared memory (S-DSM) systems to extend this programming model into message-based environments [1, 17, 21]. Object-based systems can of course be implemented on top of shared memory, but the lower-level interface suffices for many applications.

Unfortunately, despite some 15 years of research, S-DSM remains for the most part a laboratory curiosity. The explanation, we believe, lies with the choice of application domain. The S-DSM community has placed most of its emphasis on traditional parallel programming: scientific applications running on low-latency networks of homogeneous machines, with a single protection domain, usually a single programming language, and a primitive (everything lives or dies together) failure model. Within this narrow framework, S-DSM systems provide an upward migration path for applications originally developed for small cache-coherent multiprocessors, but the resulting performance on clusters of up to a dozen nodes (a few dozen processors), while good, does not lead us to believe that S-DSM will scale sufficiently to be competitive with hand-written message-passing code for large-scale parallel computing.

As an abstract concept, we believe that shared memory has more to offer to distributed computing than it does to parallel computing. For the sake of availability, scalability, latency, and fault tolerance, most distributed applications cache information at multiple sites. To maintain these copies in the face of distributed updates, programmers typically resort to ad-hoc messaging protocols that embody the coherence and consistency requirements of the application at hand. The code devoted to these protocols often accounts for a significant fraction of overall application size and complexity, and this fraction is likely to increase. We see the management of shared state as ripe for automation.

Distributed applications should be able to share program variables as easily as people share web pages. Like hardware cache coherence, or S-DSM within clusters, a system for distributed shared state should provide a uniform name space, and should maintain coherence and consistency automatically. Unlike these more tightly coupled systems, however, it should address concerns unique to wide area distribution:

- Names should be machine-independent, but cached copies should be accessed with ordinary reads and writes.

- Sharing should work across a wide variety of programming languages and hardware architectures.
- Shared data should be persistent, outliving individual executions of sharing applications.
- Coherence and consistency models should match the (generally very relaxed) requirements of applications. From a performance perspective, users should not be forced to pay for unnecessary communication; from a semantic perspective, they should be able to control the points at which updates become visible to others.
- Important optimizations, of the sort embodied by hand-tuned ad-hoc coherence protocols, should be explicitly supported, in a form that allows the user to specify high-level requirements rather than low-level implementations.

By replacing ad-hoc protocols, we believe that automatic distributed shared state can dramatically simplify the construction of many distributed applications. Java and C# programmers routinely accept the overhead of byte code interpretation in order to obtain the conceptual advantages of portability, extensibility, and mobile code. Similarly, we believe that many developers would be willing to accept a modest performance overhead for the conceptual advantages of shared state, if it were simple, reliable, and portable across languages and platforms. By incorporating optimizations that are often too difficult to implement by hand, automatic distributed shared state may even *improve* performance in important cases.

We see shared state as entirely compatible with programming models based on remote invocation. In an RPC/RMI-based program, shared state serves to

- eliminate invocations devoted to maintaining the coherence and consistency of cached data;
- support genuine reference parameters in RPC calls, eliminating the need to pass large structures repeatedly by value, or to recursively expand pointer-rich data structures using deep-copy parameter modes;
- reduce the number of trivial invocations used simply to put or get data.

These observations are not new. Systems such as Emerald [13], Amber [4], and PerDiS have long employed shared state in support of remote invocation in homogeneous object-oriented systems. Clouds [8] integrated S-DSM into the operating system kernel in order to support thread migration for remote invocation. Working in the opposite direction, Koch and Fowler [14] integrated message passing into the coherence model of the TreadMarks S-DSM system. Kono et al. [15] support reference parameters and

caching of remote data during individual remote invocations, but with a restricted type system, and with no provision for coherence across calls.

RPC systems have long supported automatic deep-copy transmission of structured data among heterogeneous languages and machine architectures [12, 25], and modern standards such as XML provide a language-independent notation for structured data. To the best of our knowledge, however, no one to date has automated the typesafe sharing of structured data in its *internal* (in-memory) form across multiple languages and platforms, or optimized that sharing for distributed applications. We propose to do so.

Over the past three years we have developed the *InterWeave* system to manage distributed shared state. InterWeave allows programs written in multiple languages to map persistent shared *segments* into their address space, regardless of Internet address or machine type, and to access the data in those segments transparently and efficiently once mapped. InterWeave currently runs on Alpha, Sparc, x86, MIPS, and Power series processors, under Tru64, Solaris, Linux, Irix, AIX, and Windows NT (XP). Currently supported languages are C, C++, Java, Fortran 77, and Fortran 90. Driving applications include datamining, intelligent distributed environments, and scientific visualization.

The following section provides a brief overview of the InterWeave programming model, touching on naming and persistence, language and machine heterogeneity, user-visible optimizations, remaining open questions, and prototype performance. Section 3 describes our current status and plans, including application development.

2. InterWeave

In keeping with the goals discussed in Section 1, InterWeave aims to support simple, safe, and efficient sharing of state among distributed applications. Within each client process, this state takes the form of ordinary global and heap-allocated variables, with layout in memory appropriate to the local programming language and machine architecture. Unlike shared variables in a cache-coherent multiprocessor, however, data shared in InterWeave have uniform, Internet-wide names (based on URLs), and outlive the executions of individual processes. They are grouped together into versioned *segments*, protected by reader-writer locks. The underlying implementation incorporates a wide variety of performance-enhancing optimizations [5, 6, 22]. Two higher-level optimizations—specification of coherence/consistency semantics and refinement of segment *views* [7]—are available to the user for additional performance tuning.

Each segment in InterWeave takes the form of a self-descriptive heap within which programs allocate strongly typed *blocks* of memory at run time. (Blocks may also be

statically allocated, and associated with named global variables. This mechanism is essential for Fortran 77, but can be used in other languages as well.) Every segment is specified by an Internet URL. The blocks within a segment are numbered and optionally named. By concatenating the segment URL with a block name or number and optional offset (delimited by pound signs), we obtain a *machine-independent pointer (MIP)*: “foo.org/path#block#offset”. To accommodate heterogeneous data formats, offsets are measured in primitive data units—characters, integers, floats, etc.—rather than in bytes.

Every segment is managed by an InterWeave server associated with the segment’s URL. Different segments may be managed by different servers. Client programs obtain access to segments by making calls to the InterWeave library. Assuming appropriate access rights, the `IW_open_segment()` call communicates with the appropriate server to open an existing segment or to create a new one if the segment does not yet exist. The call returns an opaque *handle* that can be passed as the initial argument in calls to `IW_malloc()`.

As in multi-language RPC systems, the types of shared data in InterWeave must be declared in an interface description language (IDL). The InterWeave IDL compiler translates these declarations into the appropriate programming language(s). It also creates initialized *type descriptors* that specify the layout of the types on the specified machine. The descriptors must be registered with the InterWeave library prior to being used, and are passed as the second argument in calls to `IW_malloc()`. These conventions allow the library to translate to and from a machine-independent wire format, ensuring that each type will have the appropriate machine-specific byte order, alignment, etc. in locally cached copies of segments [22].

Synchronization takes the form of reader-writer locks. A process must hold a writer lock on a segment in order to allocate, free, or modify blocks. The lock routines take a segment handle as parameter.

Given a pointer to a block in an InterWeave segment, or to data within such a block, a process can create a corresponding MIP:

```
IW_mip_t m = IW_ptr_to_mip(p);
```

This MIP can then be passed to another process through a message, a file, or a parameter of a remote procedure. Given appropriate access rights, the other process can convert back to a machine-specific pointer:

```
my_type *p = (my_type*)IW_mip_to_ptr(m);
```

The `IW_mip_to_ptr()` call reserves space for the specified segment if it is not already locally cached (communicating with the server if necessary to obtain layout information for the specified block), and returns a local machine address. Actual data for the segment will not be

copied into the local machine unless and until the segment is locked.

It should be emphasized that `IW_mip_to_ptr()` is primarily a bootstrapping mechanism. Once a process has one pointer into a data structure, any data reachable from that pointer can be directly accessed in the same way as local data, even if embedded pointers refer to data in other segments. InterWeave’s pointer-swizzling [24] and data-conversion mechanisms ensure that such pointers will be valid local machine addresses. It remains the programmer’s responsibility to ensure that segments are accessed only under the protection of reader-writer locks.

When modified by clients, InterWeave segments move over time through a series of internally consistent states. When a process first locks a shared segment (for either read or write), the InterWeave library obtains a copy from the segment’s server. At each subsequent read-lock acquisition, the library checks to see whether the local copy of the segment is “recent enough” to use [6]. If not, it obtains an update from the server. Twin and diff operations [2], extended to accommodate heterogeneous data formats [22], allow the implementation to perform an update (or to deliver changes to the server at the time of a write lock release) in time proportional to the fraction of the data that has changed.

Further performance optimizations can be effected by the InterWeave user through the use of user-specified coherence models. This is in keeping with the goal of allowing the user to specify high-level requirements rather than deal with low-level implementation issues. Coherence models can be specified in both the *temporal* and the *spatial* domain. In the *temporal* domain, InterWeave currently supports six different definitions of “recent enough”. It is also designed in such a way that additional definitions (coherence models) can easily be added. Among the current models, *Full* coherence (the default) always obtains the most recent version of the segment; *Strict* coherence obtains the most recent version *and* excludes any concurrent writer; *Null* coherence always accepts the currently cached version, if any (the process must explicitly override the model on an individual lock acquire in order to obtain an update); *Delta* coherence [20] guarantees that the segment is no more than x versions out-of-date; *Temporal* coherence guarantees that it is no more than x time units out of date; and *Diff-based* coherence guarantees that no more than $x\%$ of the primitive data elements in the segment are out of date. In all cases, x can be specified dynamically by the process. All coherence models other than *Strict* allow a process to hold a read lock on a segment even when a writer is in the process of creating a new version. The *Delta*, *Temporal*, and *Diff-based* models are reminiscent of the *continuous consistency* mechanism developed independently by the TACT group at Duke [26].

In the *spatial* domain, InterWeave provides a *view* mechanism [7] that allows a process to indicate that it is inter-

ested in only a portion of the data in a segment. Under normal circumstances, InterWeave clients cache entire copies of segments. If a client requires only a portion of the segment, however, it can specify that portion as its view. Subsequent updates to other parts of the segment will not be propagated to the local copy.

Views can be specified as either a set of (local) address ranges or as the memory reachable (recursively) through a set of pointers. We have used address-range views in a distributed object recognition system to allow a process to inspect (and pay for) only that portion of an image in which it expects to find what it wants. We have used pointer-based views in a remote datamining benchmark to cover the portion of an itemset lattice below a given node.

2.1. Open Issues

As in any system that allows a process to lock resources, we must decide whether it is acceptable to request a lock that is already held. InterWeave currently associates a counter with each segment, which it increments when a lock is re-requested. A corresponding number of unlock operations are required before the lock is released back to the server. Nested requests must employ the same coherence model; inconsistent requests result in a run-time error. At present we do not allow a read lock to be acquired on top of a write lock, nor do we allow a write lock on top of a read lock, even when using full or strict coherence.

Additional issues arise under more relaxed coherence models. Suppose, for example, that process P acquires a lock on segment A using temporal coherence, and the library verifies that the currently cached copy is (just barely) recent enough to use. If P subsequently requests the same lock on A, a repeat of the “recent enough” check might fail. Rather than introduce the possibility of a run-time error, the current version of InterWeave simply increments the lock counter and continues.

More subtly, an attempt to acquire a lock may fail due to inter-segment inconsistency. Suppose, for example, that process P has acquired a read lock on segment A, and that the InterWeave library determined at the time of the acquire that the currently cached copy of A, though not completely up-to-date, was recent enough to use. Suppose then that P attempts to acquire a lock on segment B, which is not yet locally cached. The library will contact B’s server to obtain a current copy. If that copy was created using information from a more recent version of A than the one currently in use at P, a consistency violation has occurred. Users can disable this consistency check if they know it is safe to do so, but under normal circumstances the attempt to lock B must fail. The problem is exacerbated by the fact that the information required to track consistency (which segment versions depend on which?) is unbounded. InterWeave hashes

this information in a way that is guaranteed to catch all true consistency violations, but introduces the possibility of spurious apparent violations [6].

As a partial solution to the problems of deadlock and of both real and spurious consistency violations, the InterWeave API currently provides a mechanism to acquire a *set* of locks together, allowing the implementation to obtain mutually consistent copies of the segments as a single operation. This mechanism, however, can be used only when a process is able to determine in advance the full set of locks it will need in order to complete an operation. If data inspected under one lock may determine which of several other locks may be required, deadlocks and inconsistency can still arise.

As a more complete solution, especially for applications that rely on remote invocation, we are considering a transaction-based API. Aborts and retries could then be used to recover from deadlock or inconsistency, with automatic undo of uncommitted segment updates. We have not yet decided how ambitious to make our design: in particular, whether to support nested transactions. Similarly, we have not yet decided whether or how to make inconsistent internal states of a segment visible to a remote invocation's server when the caller holds a write lock.

2.2. Performance

In previous papers [6, 7, 22] we report on the performance of InterWeave's coherence/consistency, data translation, and view mechanisms. In our remote datamining benchmark, transmitting diffs instead of entire segments reduces bandwidth requirements by a factor of 5. An additional factor of 4 can be obtained by using a relaxed coherence model, exploiting the ability of the application to tolerate moderately stale data without compromising correctness. (This change can be effected by modifying a single line of source code.) Yet another factor of 3–5 can be obtained in certain experiments by specifying pointer-based views.

When transmitting entire segments between clients, InterWeave achieves performance comparable to that of Sun RPC, and more than 8 times faster than object serialization in Sun's JDK 1.3.1. When only a portion of a segment has changed, InterWeave's use of diffs allows it to scale its overhead down, significantly outperforming the straightforward use of RPC/RMI. Similarly, in a remote invocation microbenchmark, reference parameter calls allow InterWeave to leverage automatic caching of data at an RPC server, thereby consuming dramatically less communication bandwidth than would be required for traditional deep-copy value parameters.

3. Status and Plans

In this position paper we have argued that future distributed systems will increasingly be oriented toward access to shared state. Mechanisms and middleware to automate coherent caching of that state can, we believe, make applications significantly easier to write and maintain. They can also increase performance in many cases by making sophisticated optimizations available to less sophisticated programs. Rochester's InterWeave system provides an effective platform for experimentation in this area.

We are actively collaborating with colleagues in our own and other departments to employ InterWeave in three principal application domains: remote visualization and steering of scientific simulations, incremental interactive data mining, and human-computer collaboration in richly instrumented physical environments. We also see InterWeave as central to the vision of *intuitive computing* mentioned in Section 1, for which it can provide the information sharing infrastructure.

Within our own group we are using InterWeave for research in efficient data dissemination across the Internet, and in the partitioning of applications across mobile and wired platforms. We are also working to improve InterWeave's scalability, fault tolerance, and location independence, and to provide a shared-state infrastructure for increasingly popular peer-to-peer applications.

References

- [1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, Feb. 1996.
- [2] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 152–164, Pacific Grove, CA, Oct. 1991.
- [3] P. Cederqvist and others. Version Management with CVS, Signum Support AB, 1993. Available at <http://www.cvshome.org/docs/manual/>.
- [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 *Proc. of the 12th ACM Symp. on Operating Systems Principles*, pages 147–158, Litchfield Park, AZ, Dec. 1989.
- [5] D. Chen, C. Tang, S. Dwarkadas, and M. L. Scott. JVM for a Heterogeneous Shared Memory System. In *Proc. of the Workshop on Caching, Coherence, and Consistency (WC3 '02)*, New York, NY, June 2002. Held in conjunction with the *16th ACM Intl. Conf. on Supercomputing*.
- [6] D. Chen, C. Tang, X. Chen, S. Dwarkadas, and M. L. Scott. Multi-level Shared State for Distributed Systems. In *Proc.*

- of the 2002 Intl. Conf. on Parallel Processing, pages 131–140, Vancouver, BC, Canada, Aug. 2002.
- [7] D. Chen, C. Tang, B. Sanders, S. Dwarkadas, and M. L. Scott. Exploiting High-level Coherence Information to Optimize Distributed Shared State. In *Proc. of the 9th ACM Symp. on Principles and Practice of Parallel Programming*, San Diego, CA, June 2003.
- [8] P. Dasgupta, R. Ananthanarayan, S. Menon, A. Mohindra, and R. Chen. Distributed Programming with Objects and Threads in the Clouds System. *Computing Systems*, 4(3):243–275, Summer 1991.
- [9] P. Ferreira, M. Shapiro, X. Blondel, O. Fambon, J. Garcia, S. Kloosterman, N. Richer, M. Roberts, F. Sandakly, G. Coulouris, J. Dollimore, P. Guedes, D. Hagimont, and S. Krakowiak. PerDiS: Design, Implementaiton, and Use of a PERsistent DIstributed Store. Research Report 3525, INRIA, Rocquencourt, France, Oct. 1998.
- [10] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Architecture*. Morgan Kaufmann Publishers, 1998.
- [11] A. S. Grimshaw, W. A. Wulf, and the Legion team. The Legion Vision of a Worldwide Virtual Computer. *Comm. of the ACM*, 40(1):39–45, Jan. 1997.
- [12] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Trans. on Programming Languages and Systems*, 4(4):527–551, Oct. 1982.
- [13] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Trans. on Computer Systems*, 6(1):109–133, Feb. 1988. Originally presented at the 11th ACM Symp. on Operating Systems Principles, Nov. 1987.
- [14] P. T. Koch and R. Fowler. Message-Driven Relaxed Consistency in a Software Distributed Shared Memory. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, pages 75–85, Monterey, CA, Nov. 1994.
- [15] K. Kono, K. Kato, and T. Masuda. Smart Remote Procedure Calls: Transparent Treatment of Remote Pointers. In *Proc. of the 14th Intl. Conf. on Distributed Computing Systems*, pages 142–151, Poznan, Poland, June 1994.
- [16] J. Kubiawicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, Cambridge, MA, Nov. 2000.
- [17] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, Nov. 1989.
- [18] Object Management Group, Inc. The Common Object Request Broker: Architecture and Specification, Revision 2.0. Framingham, MA, July 1996.
- [19] M. Satyanarayanan and M. Spasojevic. AFS and the Web: Competitors or Collaborators? *ACM SIGOPS Operating Systems Review*, 31(1):18–23, Jan. 1997.
- [20] A. Singla, U. Ramachandran, and J. Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *Proc. of the 9th Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 211–220, Newport, RI, June 1997.
- [21] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Konthanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 170–183, St. Malo, France, Oct. 1997.
- [22] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Efficient Distributed Shared State for Heterogeneous Machine Architectures. In *Proc. of the 23rd Intl. Conf. on Distributed Computing Systems*, Providence, RI, May 2003.
- [23] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A Wide-Area Distributed System. In *IEEE Concurrency*, pages 70–78, Jan.-Mar. 1999.
- [24] P. R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *Proc. of the Intl. Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, Sept. 1992.
- [25] Xerox Corporation. Courier: The Remote Procedure Call Protocol. Technical Report X SIS 038112, Dec. 1981.
- [26] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, pages 305–318, San Diego, CA, Oct. 2000.