PerDiS — a Persistent Distributed Store for Cooperative Applications

Marc Shapiro, Sytse Kloosterman, Fabio Riccardi

PerDiS – a Persistent Distributed Store for Cooperative Applications

Marc Shapiro, Sytse Kloosterman, Fabio Riccardi

INRIA – Project SOR, BP. 105, 78153 Le Chesnay Cedex, France
http://www-sor.inria.fr/

February 21, 1997

The PerDiS platform deploys a novel technology for the distributed sharing of information: a persistent distributed store, which provides a shared memory abstraction which is tranparently made persistent thanks to distributed garbage collection. This data sharing platform supports interactive concurrent engineering applications in a virtual enterprise. In this paper we give an overview of the project, its motivation and the PerDiS platform architecture. We also address some expected problems and related work.

1 Introduction

The PerDiS[1] project aims at studying and building a persistent distributed store (PDS) for cooperative concurrent engineering (CE) applications. These applications require a highly efficient sharing mechanism, providing immediate access to data, as opposed to remote invocation mechanisms.

To understand the PerDiS approach, consider a typical application scenario from the project's application area: the building and construction industry. Architects and engineers from different companies collaborate on a design, working at different locations, either concurrently or at different times. Tentative or alternative designs are created, tried out, abandoned. The constructors on site consult the plans, making on-the-spot modifications, which should be reflected back to the engineering offices.

This example shows the importance of data sharing as a vehicle for cooperation. The data sharing mechanisms are provided by a store, i.e. a repository of shared data. This example also shows that concurrent engineering poses various requirements on the supporting software. Since a virtual enterprise enables companies to cooperate on a joint task while still competing in other areas, security mechanisms are a requirement. End user organisations have policies for the secrecy and integrity of information used in cooperative tasks. These policies state which roles may update or view shared objects. Another characteristic of concurrent engineering is a virtual enterprise is that engineers may work on the same design in parallel. Thus mechanisms are needed that offer concurrent data access and that maintain data consistency. A design in the building and construction area typically consists of many complex objects containing many cross references. In addition, the design task itself is highly interactive. Therefore, the data sharing platform must support efficient storage and retrieval facilities for complex objects.

The current approach to data sharing in the Building and Construction industry is to snail-mail floppy disks, send electronic mail or, for the more advanced, share files over NFS [1]. This approach is clearly unsatisfactory as it is clumsy, error-prone, does not provide concurrency control, and there are no consistency guarantees.

In contrast, the PerDiS store is designed to facilitate cooperative tasks. This store is persistent, to ensure the permanence and integrity of data, and distributed, to cater for the geographical distribution of the work. It is simple to use, automatic, and efficient; it has mechanisms for tolerating faults, for security, and for supporting large-scale networks. Our architecture is called a Persistent Distributed Store (PDS). In a PDS, the communication medium is the main memory. Memory is shared between all applications, even located at different sites or running at different times. Thus, the familiar, unobtrusive, efficient, and well-typed memory API is retained.

Compared with competing technology, the PDS approach is both radically novel and gently evolutionary. It is novel because it departs from the complex APIs, unstructured data types, and specialized programming languages imposed by file systems and client-server systems, and is much simpler to use than OODBs. It is evolutionary because existing applications can be easily ported to a PDS. Programs access the PDS through the simple and familiar shared memory paradigm. A PDS preserves the semantics and typing of memory, and is known to be efficient in the common case.

[1] PerDiS is ESPRIT LTR project 22533, running from 1 Dec. 1996 until 1 Dec. 1999.

2 Comparison with the state of the art

In support of distributed shared data access, many technologies are commercially available: client-server systems; distributed file systems; object-oriented databases; and the World-Wide Web. We consider each of these in the light of two key requirements:

1. concurrent access to consistent persistent data by distributed users, and
2. a simple, easy-to-understand, powerful application programmer interface (API).

In client-server systems, as advocated by DCE [2] or CORBA [3], every data access is burdened with communication to the server, which becomes a performance and availability bottleneck. Although this architecture has been used for concurrent engineering applications, the lack of client caching makes its performance inadequate for interactive and graphical applications. The API is complicated by the need to use an interface definition language (separate from the programming language) to support remote invocation and marshalling.

Distributed file systems today are the dominant technology for the distributed sharing of information. Systems such as NFS [1] or AFS [4] support local data access at the point of use, thanks to caching. They provide some degree of fault tolerance and coherence. However they do not support the complex data types needed by modern applications such as CAD, or multimedia. The programmer's task is complicated by the need to convert between the pointer structures of objects in memory and the flattened structures on disk. Data flattening also impacts on performance. Relationships among data are usually expressed using pointers, which get lost in the flattening operation. When the data is read back in, pointers have to be inferred. For instance, a CAD application that only stores the geometrical information, has to rebuild proximity and overlapping relationships of all its graphic items.

Object-oriented databases (OODBs), such as O_2 [5], provide excellent support for complex data types. Most databases use pessimistic locking to avoid inconsistencies; when an object is locked, other clients are unable to access it, in conflict with the concurrent access requirement. Although local caching of objects improves interactive performance, previous experience shows that the performance of an OODB is inadequate for interactive applications.

The World-Wide Web [6] provides large-scale access to structured documents. Unfortunately, it provides basically read-only access, with no consistency guarantees. Web documents are large-grain entities, and its API is not adapted to Concurrent Engineering.

Although several systems that are related to PerDiS exist, PerDiS offers a unique combination of many features: distributed shared memory, persistency, replicated cached data, distributed garbage collection, failure tolerance, protection and security, scalability, performance, and a simple API.

3 Architecture

The persistent distributed store abstraction is similar to a distributed shared memory. Applications allocate their data, including complex data structures linked by references, in the PerDiS store. Data are typed; the store retains type information for each object. Type information is also contained in the store, accessible as ordinary data. Persistence is ensured by a distributed garbage collection system, based on the Larchant garbage collection algorithm [7], which implements a *persistence by reachability* (PBR) paradigm. PBR means that objects accessible from persistent roots, and only those, are stored on secondary storage. Each application runs as a separate Unix process, mapping data directly in its own address space. Access is structured into transactions; data are kept consistent across sites and across secondary storage repositories by coherence protocols.

To attain scalability, the PerDiS architecture partitions the single shared address space, conventionally offered by most Distributed Shared Memory (DSM) systems, into *clusters* of objects. Clusters are groups of related objects accessed together. This greatly simplifies the access control semantics and caching policy. A novel, scalable garbage collection technique exploits clusters to scan each one separately and concurrently on different sites [7].

Clusters are seen by application programs as files or persistent heaps: an object is allocated in a given cluster with the familiar `malloc()` operation. Clusters support persistence, by providing a directory service, which associates string identifiers with the persistence roots. The GC is aware of such persistent roots, and uses them to implement the PBR paradigm. Once an application program retrieves a root object from a cluster, it can navigate the store by dereferencing the pointers contained in the object itself.

An object may contain a reference to some object, allocated in another cluster, thereby creating a distributed network of objects. Inter-cluster references are transparently translated by the system, fully supporting the pointer semantics of low level languages such as C and C++.

To efficiently support inter cluster references in PerDiS, two tables are associated to each cluster: a stub table and a scion table. Stubs keep track of outgoing references from the cluster's objects, and scions of incoming references. Scions represent additional,

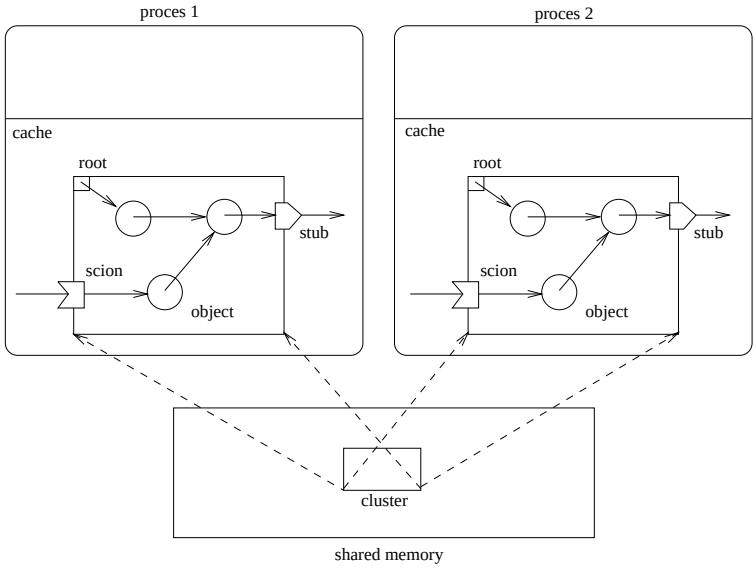

Figure 1: A cluster with references, stubs, scions and replication.

but temporary, roots of persistence for the cluster.

Clusters implement a simple protection scheme. Similarly to Unix files, they have user ownership and protection attributes.

A typical PerDiS system involves a number of sites, each site being a modern interactive workstation. Some sites may be more powerful file servers dedicated to storage and reliable backup. A site in a PerDiS system runs three types of processes: user application processes, a PerDiS daemon (PD), and a garbage collector daemon GCD. (Application processes are optional, for instance for a site that acts as a server; the PD and the GCD are mandatory.)

PerDiS is a layered system: user applications interact with the system through a simple API [8]. The API layer relies on the functionalities offered by the PerDiS user level library (ULL), which communicates with the PerDiS Daemon. The ULL is linked to the application code to form an application process. It deals with mapping data in the process addressing space, keeping track of memory allocation, and performing any necessary data conversions. It is also in charge of managing locks, and transactions of the application. The PD provides data and locks to an application process. It caches data and locks, or fetches data and locks from remote PDs, logs updates on the local log, and runs the distributed garbage collector. It also propagates updates to PDs at other sites and to disk, and sends and receives information about concurrent updates and other events (such as disconnections) affecting consistency and concurrency control.

Transaction management remains entirely inside a single ULL. When a transaction needs to read or write data or locks, it asks its local PD. Conversely, the PD may upcall the ULL to signal events that might affect data consistency, such as locks taken or commits by other transactions, or broken communication links. The ULL is allowed to cache data and locks that the application is not currently using.

The PD itself implements a per-cluster consistent distributed shared memory architecture. The PD cooperates with the GCD to perform concurrent garbage collection of the clusters currently cached on the site. Different configurations of the ULL are used by the PD and the GCD as well.

The PerDiS architecture doesn't impose any specific cluster coherence policy, which may be adapted to the specific application's requirements. Users can choose among different coherence and caching strategies for different clusters, according to the properties of their data and of their access patterns. The first implementation uses entry consistency [9] between PDS.

Communication between an application process and its PD, and between PDs, use Stub-Scion Pair Chains [7], an object request broker developed at the SOR project of INRIA. It supports garbage-collected remote object references, remote method invocation, and transparent object migration.

4 Main problems

In order to actually build the PerDiS platform, there are a number of problems that have to be tackled:

- understanding the real requirements of end-users regarding performance, functionality and interfaces,

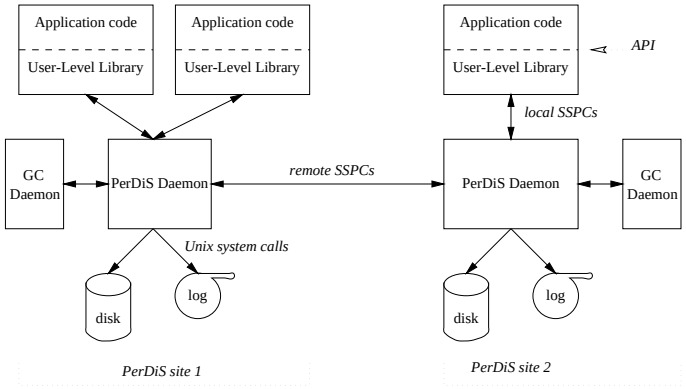

Figure 2: Decomposition into processes and sites.

- offering fine-grain control and powerful well-defined functions and tools without putting a burden on the application programmers,
- protection and security in the virtual enterprise setting; access rights, roles, insecure environments,
- fault tolerance dealing with possibly multiple faults and persistent, distributed and replicated data with tentative transactions,
- scalability,
- integration of several techniques like distributed garbage collection, concurrency control, fault tolerance and security.

5 More information

More information (such as detailed project description, involved partners, deliverables, etc.) on PerDiS can be found on its Web site:
`http://www-sor.inria.fr/perdis/`.

References

[1] R. Sandberg, D. Golberg, S. Kleiman, D. Walsh, B. Lyon, "Design and implementation of the Sun network filesystem", in *Proceedings of the Summer USENIX Conference*, pp. 119–130, June 1985.

[2] N. Leser, "The Distributed Computing Environment Naming Architecture", Open Forum, Nov. 1992, pp. 101–117.

[3] OMG, "The Common Object Request Broker: Architecture and Specification", Technical report 91.12.1 rev. 1.1, Object Management Group, 1992.

[4] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, M.J. West, "Scale and Performance in a Distributed File System", in *ACM Transactions on Computer Systems*, vol. 6, nr. 1, Feb. 1988, pp. 51–81.

[5] O. Deux and others, "The 0_2 System", in *Communications of the ACM*, vol. 34, nr. 10, Oct. 1991, pp. 34–48.

[6] T. Berners-Lee, R. Cailliau, A. Luotonen, H.F. Nielsen, A. Secret, "The World-Wide Web", in *Communications of the ACM*, vol. 37, nr. 8, Aug. 1994, pp. 76–82.

[7] M. Shapiro, P. Dickman, D. Plainfossé, "SSP chains: Robust, distributed references supporting acyclic garbage collection", research report 1799, Institut National de Recherche en Informatique et Automatique, Rocquencourt (France), November 1992. Also available as Broadcast Technical Report #1.

[8] X. Blondel, M. Shapiro, "Interfaces to Access a Persistent Distributed Shared Store in PerDiS", submitted to the 3rd Cabernet Plenary Workshop, IRISA Rennes, France, April 1997.

[9] B.N. Bershad, M.J. Zekauskas, "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors", Technical report CMU-CS-91-170, Carnegie-Mellon University, Sep. 1991.