Experience with the PerDiS large-scale data-sharing middleware

Marc Shapiro, Paulo Ferreira, Nicolas Richer

Experience with the PerDiS large-scale data-sharing middleware*

Marc Shapiro[1,2], Paulo Ferreira[3], and Nicolas Richer[2]

[1] Microsoft Research Ltd., St. George House, 1 Guildhall St., Cambridge CB2 3NH, United Kingdom; mailto:marc.shapiro@acm.org, http://www-sor.inria.fr/~shapiro/
[2] INRIA projet SOR, Domaine de Voluceau, B.P. 105, Rocquencourt, 78153 Le Chesnay Cedex, France; mailto:nicolas.richer@inria.fr, http://www-sor.inria.fr/~richer/
[3] INESC, Rua Alves Redol 9 - 6, 1000 Lisboa Cedex, Portugal; mailto:paulo.ferreira@inesc.pt, http://www.gsd.inesc.pt/

Abstract. PerDiS is a distributed persistent middleware platform, intended to ease the distributed sharing of long-term data. Its users belong to geographically-distant and non-trusting entreprises. It targets CAD applications for the building industry: data sets are large and pointer-rich; simultaneous reads and updates are supported; there is no central database; migrating legacy applications is accessible to unskilled programmers. A number of real applications have been either ported or written specifically for PerDiS.
The following design decisions were essential to the scalability and to the useability of PerDiS. Isolation (transactions) decouples users from one another. The system design provides different granularities. The programming abstraction is a fine-grain, persistent, isolated shared memory, with objects, invocations and URLs. The system mechanisms are coarse-grained, loosely-coupled and optimistic. Fine-grain application entities are encapsulated into coarse-grain system entities, respectively clusters, domains, transactions, and projects.

1 Introduction

PerDiS provides a new technology for sharing over the Internet, the Persistent Distributed Store. PerDiS enables application programmers to use shared, complex objects in a distributed environment. PerDiS is a middleware layer, designed for use in cooperative CAD applications across long distances and across organisational boundaries. Some of the requirements are to ease the porting of a legacy application by programmers unskilled in distributed systems; to support large and pointer-rich data sets; to support simultaneous reads and updates; and support multiple, geographically-distant storage sites. Interfacing to PerDiS is easy and natural: an application allocates its data in (the PerDiS-managed) memory.

* Published in Ninth International Workshop on Persistent Object Systems, Lillehammer (Norway), 5–7 September 2000. http://www-ppg.dcs.st-and.ac.uk/Conferences/POS9/

Since the major PerDiS abstraction is just memory, sharing some object is just a matter of setting a pointer to it.

Objects automatically become persistent and shared over the network. Transactions isolate applications from one another. Caching and replication techniques ensure that data is accessed efficiently and rapidly. Automatic persistence and garbage collection take away many of the headaches of sharing. Objects are protected from unwanted access or tampering by distributed access control and encryption.

Although our primary target is the building and construction industry, the technology should be applicable to other application domains where distributed access to complex, mutable data is needed.

All the results of PerDiS are available on our Web site http://www.perdis.esprit.ec.org/, in accordance with our open source policy.

2 Requirements

2.1 Cooperative work in Building and Construction

A Building and Construction (B&C) project is represented by an object for each wall, window, door, fitting, etc. The size of an object is of the order of 1–10 Kbytes. Objects are connected by pointers (e.g., a wall points to all its windows, doors, adjacent walls, floor and ceiling; and vice-versa). All told the data size of a reasonable building is in the order of tens of megabytes.

A B&C project often involves a "Virtual Entreprise" (VE), a temporary alliance between specialists of different domains (such as architects, structural engineers, heating or electricity specialists) from geographically-distant and mutually untrusting companies.

Previous attempts at cooperative CAD tools by our partners [2] were based on remote object invocation [8]: a client application running on a workstation invokes objects, stored in a remote server, through remote references. With fine-grain CAD objects this architecture results in abysmal performance[1] (especially over a WAN in a VE) unless applications are completely re-engineered.

Other attempts have used object-oriented databases, whose interface is appropriate to this application area. However OODBs are are not designed to be used across slow connections or across trust boundaries. Furthermore, each company in the VE is likely to have their own storage system; therefore a central database cannot be assumed.

2.2 B&C requirements

The foremost requirement is for a middleware layer that insulates application programmers from the complexities of distributed programming. However some network imperfections cannot be totally masked; much of our design effort has

[1] See Section 5.1 for a performance comparison between remote invocation and PerDiS on CAD data.

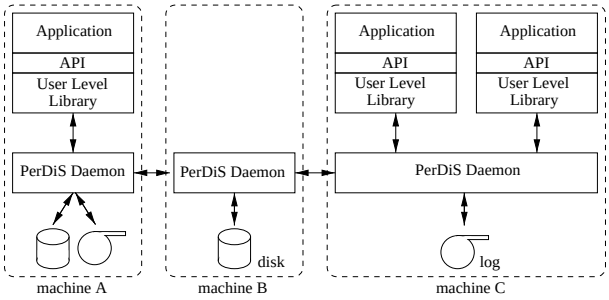


Fig. 1. PerDiS platform design

been to make sure they only appear at clear, intuitive and coarse-grain boundaries.

This middleware should make it easy to share the common data of a building project. PerDiS fulfills these requirements by offering the simplest, most familiar API: a network-wide, persistent shared memory.

In B&C, collaboration follows an essentially sequential workflow between workgroups. People within a workgroup exchange a high volume of data with each other. There is a high degree of temporal and spatial locality within a workgroup. There is also some real concurrency, including write conflicts, which cannot be ignored; for instance working on alternative designs in parallel is common practice. Each of the workgroups is small, trusting, and typically located connected by a fast LAN. In contrast, different workgroups often belong to geographically-distant and mutually-distrusting companies, connected by slow, unreliable and sometimes intermittent WANs. These considerations justify the concept of a *domain*, explained in Section 3.1.

The data stored in the platform should inter-operate with different applications on different platforms. This requirement was satisfied in the PerDiS project by implementing an industry standard interface called SDAI. Implementing SDAI was made very easy thanks to the shared memory abstraction [13].

3 PerDiS basics

The PerDiS architecture is illustrated in Figure 1. The different processes isolate crucial platform functions from applications, and applications from one another. It uses peer-to-peer cooperative caching for best performance.

A node runs a single *PerDiS Dæmon* (PD), and any number of application processes. Applications interact with PerDiS through an API layer, which interfaces to the *User Level Library* (ULL).

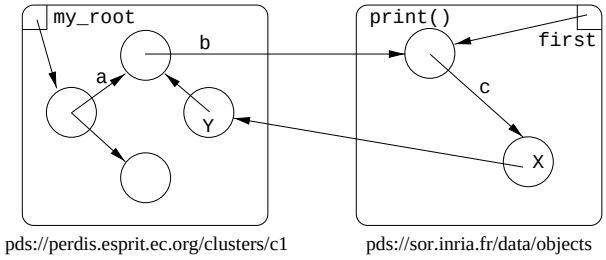


Fig. 2. Clusters, objects and remote references.

3.1 Domains

We make an essential distinction between two modes of operation: in-domain (i.e., within a workgroup, on a LAN) and inter-domain (e.g., between workgroups, or over a WAN).

A domain is a set of well-connected and mutually-trusting workstations. They share data using coherent caching and strong transactional semantics. A domain is designed for a fast LAN connection.

In contrast, the protocols between domains are designed for loosely-coupled interaction. Workgroups that distrust each other, those that interact infrequently, or those separated by a poor WAN connection, should be in separate domains.

Each domain has a *gateway* node responsible for interacting with off-domain servers. This gateway caches remote clusters and provides access to local clusters to remote domains.

3.2 Objects and Clusters

A PerDiS object is a sequence of bytes representing some data structure. Objects are opaque to the PerDiS platform, although the system stores a type descriptor for every object, and although its garbage collector is capable of following pointers in an object.

The memory is partitioned into coarse-grain structures called clusters. A cluster is a physical grouping of logically related objects. A cluster has a variable and unlimited size; clusters are disjoint.

When any object in a cluster is accessed at some site, the whole cluster is cached, supporting very fast, local access to all the objects of the cluster. A cluster is cached and possibly stored at all sites where it is in use, but each cluster has a designated *home site* storing its authoritative version. This was an application requirement for legal reasons.

An object in some cluster may point to some other object, either in the same cluster or in another one. The application program navigates the object graph from a root. See for instance Figure 2. A program might navigate from the root point in the leftmost cluster, executing for instance `my_root->a->b->print()`. This implicitly loads the corresponding cluster.

3.3 Caching and Replication

A PerDiS application always accesses the objects it needs in its local memory. This makes it easy for programmers unskilled in distributed systems to share information. Since furthermore data is cached in large chunks, this eliminates the remote access bottleneck. The system caches data lazily, i.e., only when an application accesses it. Caches at different sites cooperate to fetch the data that an application needs while minimising input-output operations.

The PerDiS API allows access at whatever granularity is most appropriate. (The default is to fetch a whole page into application memory on a page fault.) The application programmer may choose to access a small object at a time for greatest concurrency, or to access a whole range of adjacent objects in one go, for consistency. However the system mechanisms are coarse-grain: when an application requests access to a single byte, the whole enclosing cluster is opened and brought into the cache of the corresponding PD.

3.4 Transactions

PerDiS supports a broad range of transactional facilities, motivated by the application domain of cooperative engineering.

PerDiS provides different flavours of transactions: (i) classical pessimistic ACID (Atomic, Consistent, Isolated, and Durable); (ii) optimistic ACID; and (iii) check-out/check-in. Pessimistic concurrency control locks data while accessed, and holds on to a lock until the transaction terminates. Optimistic concurrency control will access data concurrently without locking; conflicts are detected through data versioning.

Check-out/check-in transactions control inter-domain sharing. (Domains were defined in Section 3.1.) When domain A needs data whose home site is in domain B, a *gateway* in domain A takes copies of the data for faster service and availability. The unit of inter-domain sharing is a *project*, a predefined set of clusters to be accessed together. A check-out transaction brings a project from its remote domain(s) into the current domain; it is cached on the local gateway. The clusters of the project can be manipulated and shared within the domain using the usual pessimistic or optimistic transactions. Finally a check-out transaction stores the updated clusters of the project in their respective home domains. The system detects at this point any potential conflicts. Currently it is up to the application to repair such conflicts.

3.5 Security

Protecting data in VEs is important, since partners in one project may be in competition for another one. Our security mechanisms [7] provide both access control and secure communication. Each cluster has an access control list indicating the access rights assigned a to a user's role within a task. Secure communication uses public key schemes for signed data access requests, shared keys for encryption and a combination of the two for authentication of message originators.

4 Distributed memory management

4.1 Distributed Memory Management

The memory management (MM) state is recorded in data structures such as free lists, object tables, roots, etc., collectively called the MM *meta-data*. Meta-data constitutes a shared resource, which in a distributed system faces concurrency and consistency issues.

To avoid locking meta-data, we propose a new transaction model, called *system transactions*.

A system transaction is a sub-transaction (of a user transaction) that takes into account the semantics of its operation. The operation can either be applied immediately and undone if the transaction aborts, or delayed and redone when the transaction commits.

Consider for instance a top-level (application) transaction that creates a new cluster. This operation is executed in its local addressing space and within a system transaction. Only at commit time is the cluster is inserted in the global cluster list, under a short-term exclusive lock. If the transaction aborts, the cluster is never inserted in the list, and never becomes visible to other transactions. This delayed operation is safe because the order in the list is immaterial to the semantics of clusters. Note that this technique still works in a disconnected environment, where the insertions can safely be delayed until the network is reconnected.

In contrast, segment reservation cannot be delayed because two different segments must not have the same address. Creating a new segment reserves its address immediately (under a short locks to avoid blocking other transactions) and records it in an undo log. If the transaction aborts, the undo log cancels its reservations.

In both cases, the semantics of the operations in the system transaction is well-known; this technique cannot be generalised to arbitrary objects.

4.2 Distributed Garbage Collection (DGC)

Persistence in PerDiS is based on reachability [3]. To decide whether an object is reachable is a difficult problem in a distributed and replicated environment. We use the Larchant distributed GC model and algorithm [9, 14]. A DGC is modeled as a per-site constructive component, which identifies new pointers, a per-site destructive component, which identifies unreachable objects, and a distribution component, which ensures consistency of GC meta-data. Larchant posits five ordering rules that enforce a safe global ordering between mutation, data coherence, and GC components, such that a destructive GC never misses a reachable object. These rules are general enough to ensure safety of any distributed GC, whatever consistency model is used.

PerDiS uses a novel algorithm for its distribution component [5, 6], based on Goldberg's hierarchical weighted reference counting [1]. This is a variation on

Weighted Reference Counting [4, 16] where each node in the diffusion tree keeps an independent count of its children so to avoid weight underflow.

The Larchant rules impose that the constructive GC run before events that act on mutator updates. The simplest way to enforce this is to run it at transaction commit. However scanning the updates is expensive and introduces a bottleneck into the commit path. To diminish the impact on performance, we are currently moving the constructive GC out of the commit path, at the expense of more complex scheduling. It does not slow down the transaction commit anymore, and can run in parallel with local mutators, which are allowed to access the newly-committed data.

However, the constructive GC must finish before either the destructive GC or the coherence engine can access the newly-committed data. This impacts mutators on remote sites, which cannot access committed data immediately. In effect we have moved the (unavoidable) constructive-GC bottleneck from the transaction commit path to the remote communication path. The advantage is that local applications are not slowed down by it any more.

5 Application experience

Porting an application to the PerDiS platform requires data and code conversions, which are both relatively straightforward.

Code conversion can be done in several ways. The simplest is the following: (i) embed the application in a pessimistic transaction that uses default data access (see Section 3.3), (ii) open persistent roots, (iii) replace writes into a file with commit.

These simple modifications bring many gains. In particular, there is no more need for flattening data structures, explicit disk I/O, or explicit memory management. In addition, data distribution, transactions and persistence come for free. However concurrency is limited; more concurrency can be achieved, at the cost of a bigger programming effort, using explicit data access primitives, the different transaction types, non-serialisable access, etc.

5.1 CAD Rendering Application

This section presents our experience with a CAD rendering application. This application is relatively simple, yet representative of the main loop of a CAD tool. We compare the original, stand-alone version, with a Corba and a PerDiS version.

The stand-alone version has two modules. The *read module* parses an ASCII file and instantiates the corresponding objects in memory. The *mapping module* traverses the object graph to generate a VRML view, according to object geometry (polygons) and semantics (for instance, a load-bearing wall is colored differently from a partition). The object graph contains a hierarchy of high-level objects representing projects, buildings, storeys and staircases. A storey contains

Test	Dataset	size	SPF objects	Poly-Loops
1	cstb_1	293	5 200	530
2	cstb0rdc	633	12 080	1 024
3	cstb0fon	725	12 930	1 212
4	demo225	2 031	40 780	4 091

(a) Test applications

Test	Orig	Pds1	Pds2	Cba1	Cba2
1	0.03	1.62	2.08	54.52	59.00
2	0.06	4.04	4.27	115.60	123.82
3	0.07	4.04	5.73	146.95	181.96
4	0.16	13.90	271.50	843.94	1452.11

(b) Test application execution times (s)

Table 1. Rendering application. For each test set, we provide: (a) Name of dataset; size of SPF file (KB); number of SPF objects and of polyloop objects. (b) Execution times in seconds: original stand-alone version; PerDiS and Corba port, 1 and 2 nodes.

rooms, walls, openings and floors; these are represented by low-level geometric objects such as polyloops, polygons and points.

In the Corba port, the read module is located in a server which then retains the graph in memory. The mapping module is a client that accesses objects remotely at the server. To reduce the porting effort, only four geometric classes were enabled for remote access. The port took two days. The code to access objects in the mapping module had to be completely rewritten. The functionality of this version is reduced with respect to the original since only geometric information is transmitted and the architectural semantics is lost.

In the PerDiS port, the read module runs as a transaction in one process and stores the graph in a cluster. The mapping module runs in another process and opens that cluster. The port took half a day.

The stand-alone version is approximately 4,000 lines of C++, in about 100 classes and 20 files. In the Corba version, only 5 of the classes were made remotely accessible, but 500 lines needed to be changed. In the PerDiS version, only 100 lines were changed.

Table 1 compares the three versions for various test sets and in various configurations. Compared to the stand-alone version, performance is low, but this is not surprising for a proof-of-concept platform. Compared to a remote-object system, even a mature industrial product, such as Orbix, the PerDiS approach yields much better performance.

The one-machine configuration is a Pentium Pro at 200 MHz running Windows-NT 4.0. It has 128 Mbytes of RAM and 100 Mbytes of swap space. In the two-machine configuration for Corba, the server runs on the same machine as above. The client runs on a portable with a Pentium 230 MHz processor, 64 Mbytes RAM and 75 Mbytes swap space, running Windows-NT 4.0. In the two-machine configuration for PerDiS, both processes run on the first machine, whereas its home site is on the second one.

This experience confirms our intuition that the persistent distributed store paradigm performs better than an industry-standard remote-invocation system, for data sets and algorithms that are typical of distributed VE applications.

It also confirms that porting existing code to PerDiS is straightforward and provides the benefits of sharing, distribution and persistence with very little effort.

5.2 Standard Data Access Interface in PerDiS

The Standard Data Access Interface (SDAI) is an ISO standard API that allows CAD applications to access to a common store.[2] The standard does not cover data distribution, multi-user access or security issues. In PerDiS we developed a late-binding implementation of the SDAI in C++. The code is about 50 000 lines written with Microsoft Visual C++ environment and using the MFC (Microsoft Foundation Classes) containers.

This reader part of this module loads a SPF file and populates a SDAI model. The writer exports a SDAI model in a SPF file.

The first step was to port the container classes of MFC (Clist, Cstring, Cvector, Carray, Cmap, and all the template containers). This porting took about 2 weeks, most of which was spent in understanding how MFC classes are implemented. The Cstring class was the must difficult: all its operators are overloaded to avoid duplicating the string array. To differentiate between a transient and persistent container, every element of a container is allocated in the same cluster as the container itself.

We then linked the SDAI layer with the persistent containers, with some minor modifications.

The SDAI implementation was a very positive experience. The PerDiS SDAI extends the standard, making it distributed and guaranteeing security. It showed that relatively complex, widely-used libraries such as MFC and SDAI can be ported with ease, once the original source code and the PerDiS model have been well understood. This vindicates our previous experience in porting the C++ Standard Template Library (STL). Any single-process application that uses either the STL or the MFC containers, or the SDAI, can be converted to a distributed, multi-user, persistent one simply by linking in our libraries.

5.3 Atlantis

Porting a real application to PerDiS is not always an easy task. The major difficulty occurs in legacy applications that contain their own implementation of the PerDiS facilities, such as persistence. We provide here some examples, that we discovered when porting Atlantis (a CAD tool provided by the PerDiS partner IEZ).

When an application is conceived without support for automatic persistence the programmer explicitly saves objects to disk. Later, when those objects are read from disk in order to be accessed again, new instances are created and initialized with the values read from disk. With this mechanism, the constructor is invoked each time an object is read from disk. Note that this constructor

[2] See http://www.perdis.esprit.ec.org/apf/sdai/ for more detail.

invocation does not occur in PerDiS; it is invoked only when the object is created and not when mapped into memory from disk.

In legacy applications with no automatic persistence support programers usually split objects into persistent and temporary, i.e. those that do not have to be saved on disk when the application ends. In PerDiS it is not necessary to make such a distinction. An object is saved to disk if it is reachable from the persistent root. So, if you have an application with "temporary objects", you will have to ensure that they are not reachable from the persistent root; otherwise, they will be saved to disk. This leads to decreased performance and to unnecessary disk occupation.

6 Platform assessment

A detailed assessment of the platform has been produced [12]. The main conclusions are reproduced hereafter.

6.1 Qualitative assessment

The distributed shared memory model of PerDiS is a simple paradigm that appeals to application developers. For the targeted application domain, it is much more intuitive than alternatives such as relational database or remote invocation. It frees users from explicit decisions about object persistence, and simplifies considerably the porting of legacy applications.

The distribution model of PerDiS makes it easy to distribute data. Furthermore the granularity of distribution is selectable, which makes it superior to fine-grain systems such as DCOM or Corba. At first access, a large granule is cached locally; further accesses are local, which improves execution time over Corba by a factor of 5 (for 40K objects), of memory by a factor of 15 (for 10K objects). The drawback is that currently the application must specify object groupings manually.

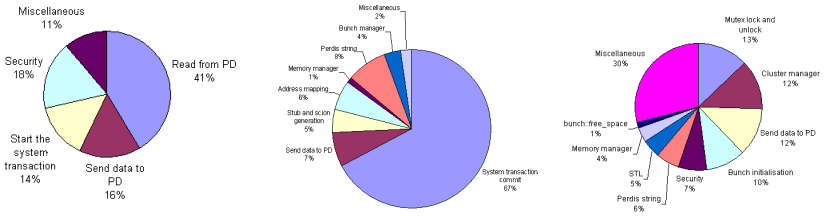
The current implementation has known limitations. Currently the system is very homogeneous, lacking a data translation layer, object relocation ("pointer swizzling" [10]), and support for the evolution of shared classes. In order to correct these shortcomings, a system for extracting type information from binary files was developed [11] but was never integrated in the platform for lack of time.

6.2 Quantitative assessment

We have done quantitative assessments of several aspects of the PerDiS platform. Some were documented in earlier sections of this paper; here we focus only on general performance.

The experimental conditions are the following: PerDiS platform version 6.1, compiled in release mode (i.e., compiler optimisations on, debugging information off). We used workstations with a Pentium II 300 MHz CPU, 128 MB of memory, and running NT 4.0. We measured and profiled the the most commonly-used

API calls. We include here the numbers for the transaction creation primitive new_transaction, the commit and abort primitive end_transaction and an end-to-end benchmark that allocates 1000 objects within a transaction. (All transactions measured are pessimistic.) For brevity we comment only on the first set of numbers.



(a) new_transaction = 72.993 ms

(b) end_transaction = 779.158 ms

(c) new_transaction + 1000*allocate_in + end_transaction = 1548.594 ms

Fig. 3. Execution time breakdowns.

Figure 3 (a) shows the execution time breakdown to start a new transaction. 57% of the time, marked "Read from PD & Send data to PD," is spent communicating with the PD through TCP/IP sockets (on the same machine). The amount of data sent is small, and sending data involves no wait for the ULL.

When the ULL reads data from the PD, it can either be actually reading data from the socket, or waiting for the PD to complete an operation before returning a value. In this latter case, it blocks in a function called wait_for_reply, whereas reading functions – the former case – do not block but actually read data from the socket. So, the time spent by the PD to perform its tasks is almost equal to the time spent by the ULL in wait_for_reply.

In all the tests we performed, the ULL spent only a few more milliseconds in wait_for_reply than in the function reading from the socket. This shows that reading from the socket accounts for half of the communication time induced by a new_transaction call. Consequently, the tasks taking most of the execution time are the ones heavily communicating with the PD.

The basic security checks and the start of the system transaction are the most expensive tasks performed in the ULL for the start of a new transaction. A glance at the code shows that these two tasks are not overwhelmingly complex; hence, most of the time is spent in communication and, more generally, in system calls.

11% of the total execution time of this primitive is spent in miscellaneous such as internal data handling (e.g. constructors and destructors, list handling...),

that account for less than 1% each. These calls are almost inherent to C++ programming, and are unavoidable.

In summary, the PerDiS platform must be evaluated for what it is, i.e., a research platform. Its main focus is testing a novel approach to sharing data in the Internet: testing new paradigms, assessing feasibility, and evaluating their cost. Performance was not the primary focus, but having demonstrated feasibility of our ideas, performance becomes essential to their acceptability. The PerDiS model is feasible, but there is much room for performance improvement. Some research issues related to performance are still open, such as memory management, and transactional support.

6.3 Lessons learned

An important issue in the design of PerDiS was to balance two conflicting requirements:

1. A simple, transparent, uniform interface, to make it easier for applications to use distribution and persistence.
2. Powerful control of, and feedback from, the distributed architecture, in order to handle the complexities and imperfections of a large distributed system.

It is not possible to maintain complete transparency in a large-scale system: failures, latency, and locality cannot (and should not) be masked. A major design issue is where such problems are made visible to the application. In a fine-grain system, the answer is "anywhere": when accessing any object, when making any call.

In contrast, a major result of PerDiS is its dual-granularity design. A PerDiS application is written in terms of the application's native, fine-grain objects, which are simply mapped into the PerDiS memory, where they can be invoked. Fine-grain entities are grouped into coarser-grain ones, which the application can control. For instance objects are grouped into clusters; an application will never encounter a problem when accessing an individual object, although it might get one when first opening its cluster. Similarly, invocations are grouped into transactions; problems occur only at the well-defined transaction boundary. Two more coarse-granularity entities are of interest to the programmer, the domain (Section 3.1) and the project (Section 3.4).

The PerDiS architecture distinguishes between local (within a domain) and inter-domain operation. As argued earlier, this answers both application requirements (tighter coupling within a work group) and system constraints: latency, security, availability, administrative are all different on a WAN. It would be a simple to extend this architecture to more hierarchy levels if there was a need.

To summarise, whereas an application deals mostly with fine-grain entities, the PerDiS platform deals with large granularities. This dual fine-grain/coarse-grain model is essential to the success and scalability of PerDiS and has a number of advantages:

1. Accessing a cluster provides implicit data pre-fetch, making future accesses much faster.

2. Expensive system calls are made infrequently.
3. A relatively small number of clusters is more manageable than an immense sea of objects.
4. Optimisation opportunities such as sharing pages between processes.
5. Adapting mechanisms to needs and costs: protocols within a domain are cheaper than across domain boundaries.
6. The failure model is simplified, because failure can occur only at coarse-grain, well-understood boundaries.

Experience confirms that our transactional design effectively combines isolation and co-operation in a wide-area loosely coupled environment. PerDiS transactions have the flexibility to adapt to a more strict and connected environment or to a typical Internet environment. Many of the transactional features introduced in PerDiS were aimed at that goal: combining optimistic and pessimistic transactions, allowing multiple locking modes, using a multi-version store which ensures data availability for remote clients and providing notifications of important transactional events.

We conclude that the main design choices have been validated. All the aspects of sharing data are combined into a uniform and simple API, including data access transparency, persistence and distribution. The approach eases the port of existing applications (within the limits of the targeted domain) and makes writing new distributed applications very easy. Some limitations remain, some that can be solved with better engineering, some that remain open research issues.

7 Conclusion

The PerDiS platform is fully functional and freely available on our Web site. It supports a number of representative applications. Our experience is that porting to PerDiS, or writing an application for PerDiS, is a relatively easy and painless way to get persistence and distribution. The current version is a research prototype; its main purpose was to demonstrate the viability of the approach. It embodies 18 man-years of research and development. Although it has a number of limitations, using it can save your own project that amount of work. Furthermore the source is freely available for others to re-use and improve.

PerDiS has a number of distinct advantages over competing approaches. It has a familiar memory interface. Its dual-granularity approach combines the best features of an intuitive, object-oriented access at the application programming language level, and simple, efficient, coarse-grain system mechanisms. In a large-scale distributed system, where network problems cannot be totally masked, one major advantage of coarse granularity is the simplified failure model: an application can be exposed to a failure only at a cluster or a transaction boundary.

Domains are another important PerDiS coarse-grain structure. They provide a mechanism for limiting the scope of expensive network operations such as cache coherence or two-phase commit. Coupled with projects they provide a way to quickly set up a new VE.

Another key factor in scalability of PerDiS is the widespread use of optimistic techniques such as optimistic and check-out/check-in transactions. Optimistic techniques are also used in other areas such as caching, security and garbage collection algorithms.

Our experience points out some interesting open issues. The direct mapping of objects into application memory is a considerable simplification, but the current model is too restrictive. Users (although happy with the shared memory abstraction) have asked us to provide different views of the same data. This would be useful for sharing data between very different applications, languages, or storage systems; it also integrates into a single coherent framework some of the needs from swizzling, security, versioning, and schema evolution. In the near future we will integrate pointer swizzling, but more general, application-specific transformations are needed, such as masking information to some users, changing measurement units, presenting the same information in different formats, or evolving class definitions. This is an area of active future research.

Another fundamental research problem is reconciliation. In a large-scale network divergence of replicas is a fact of life. Currently PerDiS detects diverging replicas and lets applications repair after the fact. Although reconciliation depends on data semantics (and indeed is not even always possible) we are trying to have the system take some of the burden and simplify the application's task [15].

Acknowledgments

We warmly thank the following members of the PerDiS project for their contribution to this paper: Xavier Blondel, George Coulouris, Jean Dollimore, Olivier Fambon, João Garcia, Sacha Krakowiak, Fadi Sandakly, and Ngoc-Khoi Tô. We also thank all those who worked on PerDiS and contributed to its success, in particular Sytse Kloosterman, Thomas Lehman, and Marcus Roberts.

References

1. Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, September 1998.
2. Virginie Amar. *Intégration des standards STEP et CORBA pour le processus d'ingénierie dans l'entreprise virtuelle.* PhD thesis, Université de Nice Sophia-Antipolis, September 1998.
3. M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.
4. D. I. Bevan. Distributed garbage collection using reference counting. In *Parallel Arch. and Lang. Europe*, pages 117–187, Eindhoven, The Netherlands, June 1987. Spring-Verlag Lecture Notes in Computer Science 259.
5. Xavier Blondel. Report on the scalability of garbage collection. Deliverable TC.1.3-B, PerDiS project, November 1999. http://www.perdis.esprit.ec.org/deliverables/docs/wpC/tc13b/.

6. Xavier Blondel. *Gestion des méta-données de la mémoire dans un environnement réparti persistant transactionnel à grande échelle : l'exemple de PerDiS*. PhD thesis, Conservatoire National des Arts et Métiers, To appear, September 2000.
7. George Coulouris, Jean Dollimore, and Marcus Roberts. Role and task-based access control in the PerDiS groupware platform. In *W. on Role-Based Access Control*, Washington DC (USA), October 1998. http://www.dcs.qmw.ac.uk/research/distrib/perdis/papers/Coulouris-RBAC9%8-final.ps.gz.
8. Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Coporation, Object Design, Inc., and SunSoft, Inc. The Common Object Request Broker: Architecture and specification. Technical Report 91-12-1, Object Management Group, Framingham MA (USA), December 1991.
9. Paulo Ferreira and Marc Shapiro. Modelling a distributed cached store for garbage collection. In *12th Euro. Conf. on Object-Oriented Prog. (ECOOP)*, Brussels (Belgium), July 1998. http://www-sor.inria.fr/publi/MDCSGC_ecoop98.html.
10. J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.
11. Alexandru Salcianu. Extraction et utilisation des informations de type pour le support des objets répartis. Mémoire de dea, École Normale Supérieure de Lyon, July 1999. http://www-sor.inria.fr/publi/EUITSOR_dea-salcianu-1999-07.html.
12. F. Sandakly, O. Fambon, M. Roberts, X. Blondel, and N. Tô. Final report on assessment of the platform and proposed improvements. Deliverable TA.3-A, PerDiS project, March 2000. http://www.perdis.esprit.ec.org/deliverables/docs/wpA/ta3a/.
13. Fadi Sandakly, Marcus Roberts, and Thomas Lehmann. Report on experience programming over IPF. Deliverable TA.2.2-D, PerDiS Project, March 2000. http://www.perdis.esprit.ec.org/deliverables/docs/wpA/ta22d/.
14. Marc Shapiro, Fabrice Le Fessant, and Paulo Ferreira. Recent advances in distributed garbage collection. In S. Krakowiak and S. K. Shrivastava, editors, *Recent Advances in Distributed Systems*, volume 1752 of *Lecture Notes in Computer Science*, chapter 5, pages 104–126. Springer-Verlag, February 2000. http://www-sor.inria.fr/publi/RAIDGC_lncs1752.html.
15. Marc Shapiro, Antony Rowstron, and Anne-Marie Kermarrec. Application-independent reconciliation for nomadic applications. In *Proc. SIGOPS European Workshop: "Beyond the PC: New Challenges for the Operating System"*, Kolding (Denmark), September 2000. ACM SIGOPS. http://www-sor.inria.fr/~shapiro/papers/ew2000-logmerge.html.
16. P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87—Parallel Architectures and Languages Europe*, number 259 in Lecture Notes in Computer Science, Eindhoven (the Netherlands), June 1987. Springer-Verlag.