Concurrency Control for Distributed Cooperative Engineering Applications

João Coelho Garcia
INESC ID / IST
Rua Alves Redol 9, sala 615
1000 LISBOA, PORTUGAL
+ 351 21 3100225
jog@gsd.inesc-id.pt

Paulo Ferreira
INESC ID / IST
Rua Alves Redol 9, sala 609
1000 LISBOA, PORTUGAL
+ 351 21 3100230
paulo.ferreira@inesc.pt

ABSTRACT

Distributed cooperative engineering applications require consistent and long-term sharing of large volumes of data, which may cause conflicts due to concurrent read/write operations. Therefore designing concurrency control for underlying middleware systems is a difficult issue.

Current transactional solutions, even if based on an optimistic approach, do not solve the problem because such applications access shared data for long periods of time performing a large number of read/write operations. Typically, a large set of modifications has to be discarded and this is unacceptable given the amount of work lost.

In this paper, we describe the design and implementation of concurrency control mechanisms aimed at both reducing the amount of such conflicts and supporting the consistent long-term sharing of data. The mechanism of visibility depth allows the programmer to specify the consistency of shared data w.r.t. different sets of sites. We also provide other mechanisms: private-copy that allows data to be read/written without being considered as part of a transaction and reordering transaction history to avoid transaction aborts. We evaluate these techniques on a prototypical middleware system called PerDiS and show that: (i) the concurrency control mechanisms are well adapted to support long-lived data sharing in local or wide-area networks, and (ii) performance is acceptable.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Databases

General Terms

Design, Algorithms, Performance

Keywords

Concurrency Control, Persistent Store, Cooperative Applications

1. INTRODUCTION

Distributed, cooperative engineering applications are growing in importance. An example of such applications is the computer-aided design (CAD) of buildings within a virtual enterprise (VE). A virtual enterprise (VE) is a consortium of teams from different enterprises in different geographical locations, working together for the duration of a project.

In a VE, fast and consistent access to data, despite concurrency, is a fundamental requirement. Often collaboration follows a stylized, sequential pattern. Typically, one team, usually within a LAN (local-area network), does the initial design, performing many updates during a limited period of time. Then, the design is passed along to another team possibly in a different site, which assesses a different technical aspect. They then pass their results on to another group, and so on. There is a high degree of temporal and spatial locality. There is also some real concurrency, including write conflicts, which cannot be ignored; for instance working on alternative designs in parallel is common practice.

Under these circumstances, a standard transactional model, with pessimistic transactions and two-phase locking, is not well suited to our application area because it may imply aborting very long-lived transactions. In some cases, a transaction that, according to standard semantics, would abort could in fact commit successfully with a more sophisticated transactional model.

We propose a new mechanism to avoid transaction aborts and lost work called **visibility depth**. It allows the programmer to specify the consistency of shared data w.r.t. different sets of sites. Programmers can require the system to ensure the consistency of shared data for three visibility depths:

- WAN (wide-area network): on a system-wide scale;
- LAN: in the local-area network of a VE team;
- PC: in the workstation of a team member.

This mechanism maps easily to the stylized collaboration of a VE described above and was applied to a transactional middleware system (PerDiS) [6], which was designed specifically for VEs.

In addition to the visibility depth mechanism we provide two other techniques for transactional middleware systems:

- **private-copy:** allows an application to read or write data that will not be considered as accessed when the enclosing transaction commits;
- **reordering transaction history:** changing the order with which transactions are committed so that conflicting transactions once reordered can effectively commit successfully within a new transaction history;

In this paper, we describe these techniques, how we integrated them in PerDiS and present an evaluation of the proposed techniques and of PerDiS.

SAC 2002, Madrid, Spain

2. ENVIRONMENT

Generally in VEs, members of a team work within a single LAN, while the cooperative work between teams is performed on a WAN and is less coupled.

We make a clear distinction between LANs and WANs (see Figure 1).

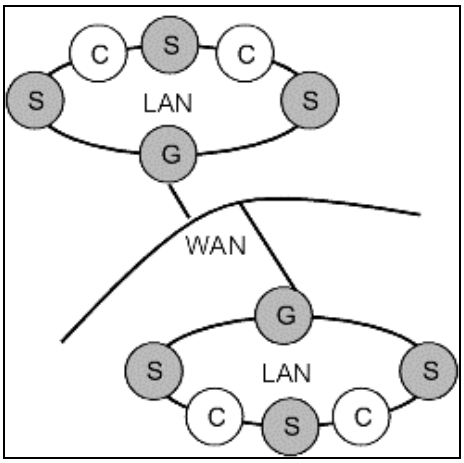

Figure 1 LAN/WAN architecture (S: PerDiS servers; G: gateways; C: client workstations)

A LAN is made of a group of sites that can act as client (application executor), data server (transaction participant) and/or transaction coordinator. All data has a homesite, which is the node where it was created. The homesite is part of the data-naming scheme and is used as a base for locating data. For each LAN, there is a special site, called the LAN's gateway that manages all interactions with sites outside the LAN.

3. VISIBILITY DEPTH

The notion of visibility depth was motivated by the need for different degrees of isolation between VE teams. As an example, consider a distributed cooperative CAD tool. Suppose that the company responsible for the pipe infrastructure in a building has a team working in that company's LAN. This team is restructuring the pipe system; thus, the team members want to see each other's work, but don't want other VE partners to see the intermediate steps of that work until all the restructuring is done. In this case, LAN visibility depth limits the visibility of the intermediate read/write operations to the company's LAN. The outcome of the changes will be shown to the rest of the consortium partners only when a WAN commit is executed.

Visibility depth allows users to define, for each transaction, the **source** from where data should be read from during a transaction and the **destination** where it should be stored into at transaction commit time. Currently, we support the following visibility depths, which provide different consistency domains:

- **WAN** - Committed updates are made available to all VE sites. It involves an actual modification of the data at its authoritative homesite.
- **LAN** - Committed updates are made visible only to other sites within a LAN and are stored in the LAN's gateway; they are not globally visible until the user performs a WAN commit.
- **PC** - Committed updates are strictly local to a workstation (where the application is running) and are not made visible to others until a WAN or LAN commit is performed on this data.

Requesting an object with visibility depth PC, LAN or WAN implies receiving a replica that is coherent w.r.t. the client workstation, LAN or WAN, respectively. The use of visibility depth implies that there will be several simultaneous versions of the same data, which may be joined if they are all brought to a global (WAN) consistency domain.

4. PRIVATE-COPY

Usually data can be accessed for reading or writing. We propose private-copy is an additional mode of requesting and accessing data. Such data is consistent when read and can be used for browsing or draft work but is not submitted at commit time. In other words, the application programmer (or its user) may access the private copy data as he wants (i.e. read or write), but that data does not belong to the transaction read or write set and therefore is not included in its validation.

For example, in CAD, a user frequently has to browse through data to get to the part of the project he is working on. This does not mean that his work depends directly on the browsed data. As a matter of fact, most probably the reading of such data has no semantic impact on the running transaction. However, if this data were marked as *read* or *written*, unnecessary conflicts might arise with other engineers updating part of the browsed data. It's worth noting that these are not real semantic conflicts between users, therefore the transaction can effectively commit. If an engineer were to change his mind and decide to really work on data that was initially requested as private-copy, the corresponding transactional lock could be upgraded to a read or write lock.

Private-copy has to be used carefully. In particular, users must not request private copies of data whose access will affect other reads or writes in the transaction; if that is the case, such data must be included in the running transaction's read (or write) set.

5. VERSIONS AND REORDERING OF TRANSACTION HISTORY

Multi-version generalized validation (MVGV) [1] is a validation and commit algorithm for distributed optimistic transactions that we have adapted to the environment and goals of PerDiS (see 4.1). Pessimistic transactions restrict concurrency by locking data conservatively; this guarantees that the transaction does not abort due to conflicts. Optimistic transactions don't lock data because they assume low contention between transactions. Consequently conflicting transactions may have to abort. MVGV manages a multi-version database where each database tuple[1] may have several versions; each version corresponds to the changes made to the tuple by a different transaction.

Transactions always have a consistent view of the database since the data versions that were valid when they began are kept at least until the transaction ends. Thus, in a multi-version store, committed transactions do not overwrite existing data versions but originate new versions instead. This has two advantages: (i) read-only transactions need never be validated or aborted because they always see consistent data and (ii) the history of transactions is kept available and can therefore be reordered to eliminate conflicts.

The distributed commit protocol is a two-phase protocol (prepare and commit) where the site where the application ran acts as coordinator, and the servers that provided data are the participants.

Additionally, during the prepare phase, while each participant validates the data updates, a site may detect a conflict that can be avoided by reordering the history of transaction instead of abort-

[1] In PerDiS, the database is a file store and each tuple is a file.

ing. When a committing transaction T1 has had its read set overwritten by another transaction T2 that committed previously, the transaction protocol proceeds to check whether the committing transaction T1 has not overwritten the read sets of any of the transactions between the committing T1 and the conflicting T2 transactions. If this check succeeds, T1 can be placed before T2 in the validation queue. A third phase in the commit protocol is used to propose this reordering to all participating sites. If it succeeds the transactions are committed in the new order; otherwise the transaction aborts.

If the user does not want to lose his updates, he can still prevent aborting the transaction by committing it at a more restricted visibility depth (LAN or PC). This way, conflicting versions are available for latter reconciliation.

6. PerDiS ARCHITECTURE

PerDiS is a distributed persistent object store. Applications running on top of PerDiS use a simple API to access persistent objects within transactions with ACID guarantees. Graphs of persistent objects are stored in files called clusters.

In PerDiS, each LAN has a site called the gateway that establishes the interface between memory sharing within the LAN and cluster sharing at WAN scale. It also holds a file cache that stores all data with LAN visibility depth and performs some cluster prefetch from remote sites. Further details on caching are out of the scope of this paper.

Transactions are done over a multi-version store managed using MVGV. PerDiS has two modes of transactional operation: one for LANs and another for WANs. This separation influences the locking, the commit method and caching. Consequently it defines the type of contention events which can be detected and notified to applications:

- Within a LAN, PerDiS sites share data coherently using a page-based distributed shared memory mechanism (DSM) [12]. At commit time, the PerDiS server at the site where the transaction ran requests all necessary locks and validates the transaction. If validation succeeds, updates are written both to DSM and to local servers' disks. Contention detection is done with page granularity given that the DSM is page-based.
- On a WAN scale, PerDiS sites share data by exchanging and caching whole files (via their gateways), file updates are logged locally and transactions are committed using MVGV. Since data is first logged locally, the commit protocol can be performed asynchronously, i.e. the application is not forced to wait for the commit result; it may proceed tentatively using the foreseeable commit result.

PerDiS includes notifications for when: other users request cluster (indicating intentions to read or write), other users update cluster and for when concurrent transactions complete. Notifications are delivered to applications using callbacks or directly via e-mail to users.

PerDiS is able of scheduling and performing asynchronously actions that were postponed due to communication problems, which sometimes occur over WANs. Asynchronous activities in PerDiS are: completing transaction commits and sending notifications. In particular, the result of asynchronous commits are conveyed to applications using notifications.

7. PerDiS IMPLEMENTATION

PerDiS is implemented as a user-level library linked with applications and a PerDiS server running at each site (see Figure 2). Normal servers perform as gateways depending on run-time configurations. In PerDiS, during transactions data is mapped in application memory using shared memory between the applications' library and the local server. At commit time updates are sent to the local server, which initiates the commit protocol (MVGV).

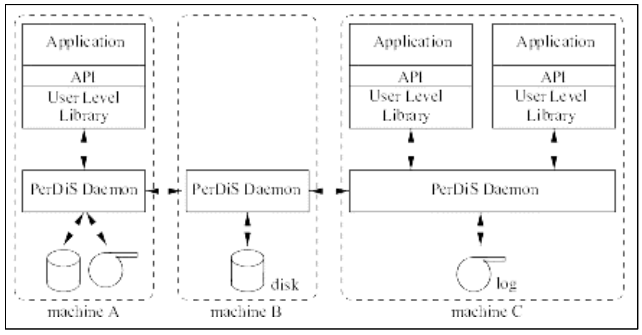

Figure 2 PerDiS architecture

In MVGV, a data structure called a validation queue is used to manage the multi-version store and represent the transaction history. Each entry in this queue represents a transaction along with the reads and writes it has performed. These entries do not correspond to pessimistic locks and don't prevent other transactions from accessing the same objects.

VQ entries are used for validating the transaction against others, for detecting conflicts and sending notifications. A transaction will have entries in the VQs of all sites from which it requests files. These entries are inserted when the first file is requested. VQ entries include the identification and intent (read, write or private-copy) of the requesting transaction, which may also used for notification purposes.

7.1 WAN Transactions

When a transaction begins, it is assigned a global start timestamp (**gst**). When it is submitted for commit, it is assigned a global commit timestamp (**gct**). Thus, each cluster is labeled with a time (or version) stamp marking the moment when it was committed. Transactions are globally ordered according to their **gct**.

This global order is established by assuring that the **gct** is a majorant of all **gct** previously assigned by the participating sites. The site where the transaction executed coordinates the distributed commit and proposes a **gct**. After validation, it is informed whether any of the participating sites had already assigned a timestamp greater than the proposed one. If that is the case, the transaction's **gct** is increased accordingly and the new value is broadcast in the two-phase commit protocol's commit message. One should note that this procedure tends to keep the timestamps of sites that interact frequently loosely synchronized.

7.2 LAN Transactions

For transactions strictly within a LAN, the commit procedure is simpler. This procedure is supported by the assumption of complete local connectivity and on the use of the DSM caching this allows. When a transaction wants to commit, all pages in its read and write sets, and the corresponding locks, are requested. The transaction is validated by checking for read/write (overwriting of its read set by others) and write/write (write set overwritten by concurrent transactions) conflicts. If the validation succeeds, and before releasing the locks, the cache is updated by writing to local memory all pages that have been fetched. The corresponding files are updated by applying their updates from the log to the files on disk using a LAN scale remote file access protocol.

8. EVALUATION

This section presents and justifies the following evaluation results: (i) how easy it is to program a distributed application on top of PerDiS, while making use of the concurrency control mechanisms presented in Section 3, and (ii) even though the prototype is non-optimized, its performance is acceptable.

8.1 Project Manager

The Project Manager (PM; Figure 3) is a demonstration application programmed on top of PerDiS to allow different users to work cooperatively over a set of application files (text, spreadsheet, etc…). We define a project as such a set of semantically related files. This notion of project allows the user to rely on the transactional PerDiS support to ensure the coherence of related files whenever one or more of them are being edited.

Users can check-out a single file or a whole project, edit the files using an external application, and later check them in to ensure their changes are visible to other users. Files may even be imported from a WWW site into a project.

This application is similar to some version control systems, such as the CVS software from Cyclic[2], in the sense that both allow check-out and check-in operations from concurrent users, and rely on external applications to edit the files. Compared to CVS, however, PM, which runs on top of PerDiS, provides transactional access to a fully distributed storage well adapted to wide-area networks. Additionally, we do not have to rely on a central server to store all the project contents as in version control systems.

Each project is described by a metadata file. Thus, whenever a file is added to a project, the project's metadata, containing the file contents and some additional information, is created (if it did not already exist). When a user checks-out files, the project manager application stores them locally (in a directory chosen by the user). Then, these files can be edited using an external application. Later files are checked-in for transactional commit. If conflicts occur at WAN visibility depth, the user can always decide to check-in with LAN or PC visibility and later reconcile [14]. When a conflict happens or the transaction commit succeeds, PerDiS automatically sends a notification to concurrent users. If a user receives a conflict notifications while editing a file, he knows that the subsequent check-in may fail; thus, he may decide to check-out that file again and integrate his changes with the newer version.

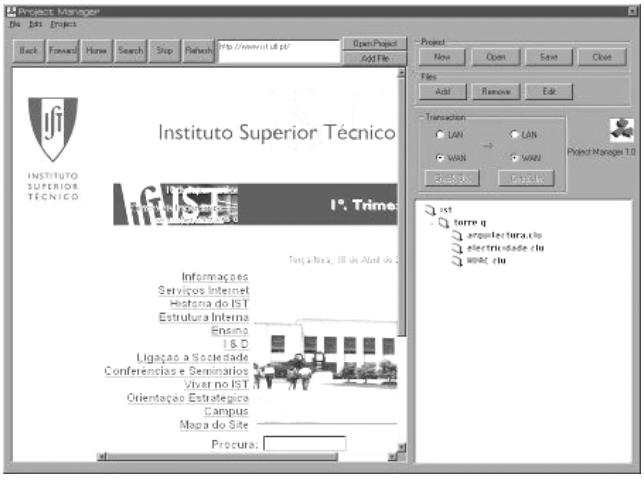

Figure 3 Project manager interface

[2] http://www.cyclic.com/

8.2 Simplicity of Programming

PerDiS has a very simple application programming interface (API). Applications only have to begin transactions and open clusters in order to obtain persistent objects. From then on, they access persistent and volatile data in the same way except for a different memory allocation primitive. When processing is finished, clusters are closed and all changes become persistent when the running transaction is committed.

As an example, Figure 4 shows the code for the PM's check-in operation, which completes the transaction initiated at check-out, and saves application files as coherent persistent objects. This task writes the files that constitute a project into the corresponding PerDiS cluster and commits this operation to the store.

Once a reference to the root object of a cluster is obtained, all the subsequent handling of local and cross-cluster references follow standard pointer semantics, except that unreachable data is prone to be garbage collected after the transaction commits [4].

All error checking has been omitted for the sake of simplicity. As it can be observed, apart from enclosing the function's contents in a transaction block and explicitly opening the root of the cluster data, the remaining code is very much like the one we find in centralized applications. This example shows that the PerDiS approach is both powerful and elegant. Thanks to cluster caching and DSM, all data manipulation is local and distribution is transparent. Local and cross-cluster references are both used as normal pointers.

```
void CheckInFile(Cluster clu,
   CString filename, transaction t) {
// open cluster and root
root = open_root( ROOT,intent_exclusive,
clu);
// open the file
CFile file(save_filename,
CFile::modeRead | CFile::typeBinary);
long file_size = file.GetLength();
BYTE *buffer = new BYTE[file_size];
// allocate the array on the cluster
root->content = allocate_array_in(
BYTE,my_clu,file_size);
f.ReadHuge(buffer, file_size);
// write the buffer's contents
// into the cluster
memcpy(root->content, buffer, file_size);
// close the file and the cluster
f.Close();
close_cluster(clu);
end_transaction(t, 0);
}
```

Figure 4 PerDiS source code for Project Manager

8.3 Performance

In this section, we evaluate the performance of the PerDiS platform based on a benchmark that closely resembles the behavior, at transaction begin (check-out) and commit (check-in), of the PM presented above.

The testing environment that we simulated as an example of a VE scenario consists of 4 PCs that make up two LAN workgroups separated by a WAN network (see Figure 5). The testing environment was the following: 4 Intel Pentium III computers with 128 MB RAM, running Windows NT 4.0. The network within the

LANs is a local 100 Mb/s Ethernet. The bandwidth between LANs is significantly reduced due to traffic generated by other users.

The data consists of two variable-sized clusters with references between them. The benchmark application takes these two clusters and opens, modifies and then commits them. In PerDiS, there is an enormous difference cluster size and actual data content. This is due only to implementation deficiencies outside of the transactional sub-system and is transparent to it. These data organization problems lead to numerous large files, which could easily be avoided through metadata reorganization. Therefore, we believe it is fair to evaluate transactional performance using the real cluster size and not its data content.

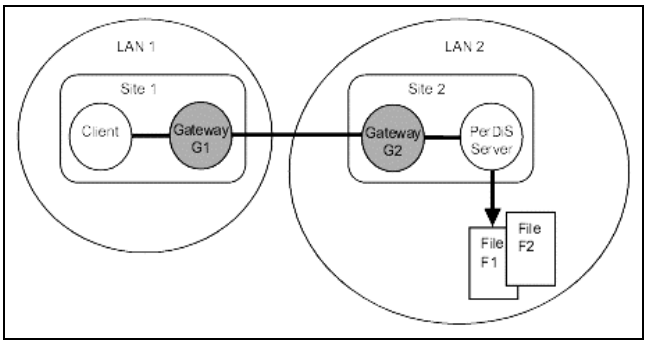

Figure 5 PerDiS testing scenario.

We compare the data's check-out and check-in duration with the aggregate cluster size and with the number of files. Figure 6 shows the performance results of the project check-out and check-in operations as the file size varies. The check-out operations presented have a high cache hit rate at the LAN gateway. Therefore, the duration of the check-out is due mainly to the coherence validation of all required files at their remote homesites and to the initializations needed to begin a transaction. As expected the check-in operation timing greatly depends on the file size. This is due to the synchronous writing on persistent storage of data and log information. However, the amount of file spaced in these tests, if adequately used, could store a large amount of data and so this scenario corresponds to a realistic commit of a large project.

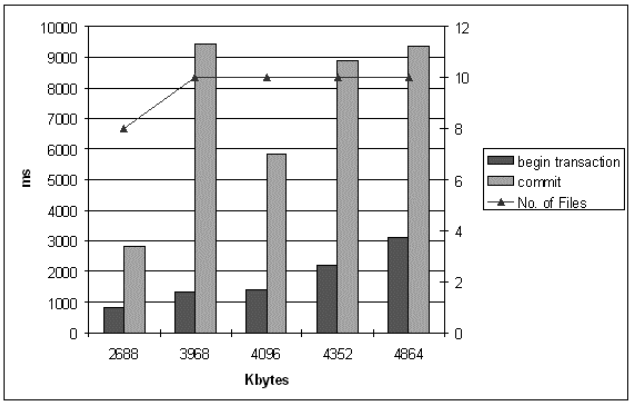

Figure 6 Check-out and check-in performance

Detailed profiling further showed that circa 75% of the overall commit time is spent sending data to the server and writing to disk. Thus, communication places a significant burden on performance at commit time. Note that, since these two activities (communication and writing to disk) are staggered, there is a large overlap between them.

9. RELATED WORK

Many middleware platforms are based on remote object invocation, using Corba [2, 5], DCOM [13] or Java RMI [17]. An application invokes objects, stored in a server, through remote references. In the CAD domain this results in abysmal performance, and server scalability problems. Remote objects are especially inappropriate in VEs, where servers may be located across a slow WAN connection. Furthermore, efficient porting of centralized CAD applications to middleware systems requires complete re-engineering. Additionally, traditional remote object systems cause a high percentage of aborts in long-term data sharing.

PerDiS can be compared to many different kinds of systems: distributed file systems, DSM systems, persistent object systems, etc.... In this paper, we focus on concurrency control especially the reduction of aborts. Several DBMS have extended standard pessimistic two-phase locking (2PL) to allow greater concurrency [11, 15]. In particular, optimistic locking protocols allow intense concurrency under workload patterns with low write contention. A client/server DBMS with inter-transactional caching at the clients requires notifications to maintain cache coherence in regard to other clients. This can be achieved by invalidating stale cache, propagating new committed data or by a dynamic choice between these techniques [7]. In cooperative environments with low write contention, invalidations perform best because they are less sensitive to usage patterns. With the transactional techniques presented for PerDiS, maintaining data consistency does not necessarily imply aborting transactions.

10. CONCLUSIONS

Distributed cooperative engineering applications require consistent and long-term sharing of large volumes of data. PerDiS supports this kind of sharing by means of a transactional system. PerDiS incorporates concurrency control mechanisms aimed at both reducing the amount of lost work due to conflicts and supporting the consistent long-term sharing of data. The mechanisms of visibility depth, private copy and reordering of transaction history, all contribute to a small number of aborts. In addition, from our experience, they provide the right support to deal with wide-area connectivity problems and to allow the kind of sharing needed in a VE environment.

The PerDiS API is very simple and similar to the ones found in centralized systems. This makes porting already existing centralized applications to PerDiS a simple task.

In summary, the contributions of this work are the following: (i) a running prototype called PerDiS, able to support both local-area and wide-area long-lived data sharing, (ii) the notion of visibility depth that is well adapted to the stylized, sequential pattern of cooperation found in a VE, as it allows the application programmer to specify where data is fetched from and where it is stored into, (iii) additional concurrency mechanisms that contribute to reduce the amount of aborts: private-copy that allows data to be read/written without being considered as part of a transaction and reordering of transaction history, and (iv) in PerDiS, local-area transactions run on top of a DSM mechanism thus providing a data-shipping approach which results both in a centralized-like application programming interface and better performance than function-shipping paradigms for long-lived transactions.

11. REFERENCES

[1] Divyakant Agrawal, Arthur J. Bernsteinm, Pankaj Gupta, and Soumitra Sengupta. Distributed optimistic concurrency control with reduced rollback. Distributed Computing, Springer-Verlag, 2:545-59, 1987.

[2] Virginie Amar. Integration des standards STEP et CORBA pour le processus d'ingenierie dans l'entreprise virtuelle. PhD thesis, Université de Nice Sophia-Antipolis, September 1998.

[3] Ken Arnold and James Gosling. The Java Programming Language. Addison-Wesley, 1996.

[4] Xavier Blondel, Paulo Ferreira, and Marc Shapiro. Implementing garbage collection in the PerDiS system. In Proc. of the Eigth International Workshop on Persistent Object Systems: Design, Implementation and Use (POS-8), 1998.

[5] 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.

[6] Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, João Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Robert, Fadi Sandakly, George Coulouris, Jean Dollimore, Paulo Guedes, Daniel Hagimont, and Sacha Krakowiak. PerDiS: design, implementation, and use of a PERsistent DIstributed Store. Recent Advances in Distributed Systems, Springer Verlag LNCS, Eds. S. Krakowiak and S.K. Shrivastava, 1752, February 2000.

[7] Michael Franklin, Michael Carey, and Miron Livny. Transactional client-server cache consistency: Alternatives and performance. ACM Transactions on Database Systems, 22(3):315-363, September 1997.

[8] Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques. Data Management Systems. Morgan Kaufmann, 1993. ISBN 1-55860-190-2.

[9] E.James Whitehead Jr. World wide web distributed authoring and versioning (webdav): An introduction. Standard View, 5(1), March 1997.

[10] E.James Whitehead Jr. and Meredith Wiggins. Webdav: IETF standard for collaborative authoring on the web. IEEE Internet Computing, September 1998.

[11] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. Communications of the ACM, 34(10):50-63, October 1991.

[12] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. ACM Transactions on Computer Systems, 7(4):321-359, November 1989.

[13] Roger Sessions. COM and DCOM: Microsoft's Vision for Distributed Objects. Wiley, 1996.

[14] Marc Shapiro, Anthony Rowstron, and Anne-Marie Kermarrec. Application-independent reconciliation for nomadic applications. In Proc. of the 9th ACM SIGOPS European Workshop, Kolding, (Denmark), September 2000.

[15] Reinhard Stroste and Herbert Eberle. Kernel service call. Technical Report 43.8701, IBM European Networking Center, January 1987.

[16] Andrew S. Tanenbaum. Computer Networks. Prentice-Hall, 1996.

[17] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. In Conference on Object-Oriented Technologies, Toronto Ontario (Canada), 1996. Usenix.

João Garcia is a teaching assistant at the Computer Science Department of IST in Lisbon (Portugal) and is a researcher at the Distributed Systems Group at INESC ID in Lisbon. His work is focused on ad-hoc networks, service location and persistent stores.

Paulo Ferreira is an assistant Professor at the Computer Science Department of IST in Lisbon and heads the Distributed Systems Group at INESC ID. His scientific interests are system support for large-scale distributed data sharing, replication and consistency protocols, distributed garbage collection, persistence by reachability, operating systems, large-scale distributed applications, internet protocols.