A Component-based Infrastructure for Customized Persistent Object Management

Luciano García-Bañuelos — Phuong-Quynh Duong — Christine Collet

LSR/IMAG Laboratory
681, rue de la Passerelle
38400 Saint Martin d’Hères, FRANCE

ABSTRACT. On this paper we present a component-based infrastructure for building customized persistent object managers. We consider that future database management systems (DBMS) will be built as a set of adaptable services. Our infrastructure is comprised of such services, and its usage is twofold. First, it can be used to provide object managers for applications where a full-fledged DBMS may be cumbersome (e.g., mobile computing). Second, it can be used as a middleware for persistent object management (e.g., on top an existing DBMS). The paper focuses on architectural principles of the infrastructure, its components, as well as the inter-component interactions and dependencies.

RÉSUMÉ. Dans cet article nous présentons une infrastructure à base de composants pour la construction de gestionnaires d’objets persistants personnalisés. Nous considérons que les futurs systèmes de gestion de bases de données (SGBD) seront construits comme un ensemble de services adaptables. Notre infrastructure comporte de tels services et peut-être utilisée pour 1) fournir des gestionnaires d’objets aux applications dans lesquelles il n’est pas nécessaire de disposer de toutes les fonctions d’un SGBD (e.g., informatique mobile), 2) offrir un intergiciel dédié à la gestion de données persistantes (e.g., au dessus d’un SGBD existant). Le document présente les principes architecturaux de l’infrastructure, ses composants logiciels, ainsi que leurs interactions et dépendances.

KEYWORDS: Database services, DBMS architectures, Component-based architectures, Persistence management, Cache management, Multi-level transactions.

MOTS-CLÉS: Services base de données, Architectures SGBD, Architectures à base de composants, Gestion de persistance, Gestion de cache, Transactions multi-niveaux.

1. Supported by the CONACyT Scholarship program of the Mexican Government
1. Introduction

This paper presents preliminary results of the NODS (Networked Open Database Services) project [COL 00]. The evolution of this work can be tracked in [Gar 01, Gar 02]. Our project vision is that DBMS should be unbundled into a set of cooperating services. With such services, we aim at constituting a framework from which system programmers may build customized data managers. This vision is shared by several other researchers (see for instance [VAS 94, SIL 97, GEP 98, HAM 99, CHA 00]). However, we can corroborate that only few active research projects are devoted to the process of revisiting DBMS architectures.

There are contexts where overheads imposed by the whole DBMS machinery cannot be afforded. This is the case of data management applications on resource constrained environments, e.g. mobile computers/devices. The single solution for scaling down DBMS to those environments seems to be building new ad hoc software. But the problem is not only down sizing.

There have been some previous attempts to develop more adaptable DBMS architectures. Most of those early works have kept the full-functional DBMS as the fundamental unit of packaging for both installation and operation.

We claim that a component-based approach to architecture DBMSs software should be taken. Thus, instead of considering extensible DBMS kernels, we propose finer-grain components. Our components provide low level services such as logging, caching, concurrency control, recovery, among others. These components can be used independently, as a subset, or altogether according to the data management requirements.

Our work takes well-known techniques for persistent object management and put them into a generalized infrastructure. The novelty of our work is the resulting architecture principles, enabling componentization. We therefore applied architectural reasoning in a systematic way to characterize component interdependencies. All these principles have been validated by building a full functional prototype.

Promoting a component-based approach is striking:

– Components can be used to build stand-alone data managers for applications with particular requirements.
– Components can be used to build database middleware.
– Systems using multiple component-based services can exploit common sub-components, improving resource sharing.

1.1. Approach

We have tackled the problem of componentizing DBMS software. However, a correct componentization of DBMS software compels a deep study because of the subtle dependencies kept by their functional constituents.
We started by working in the isolation of persistence related services from the whole DBMS machinery. The goal was to provide components to assemble persistent object managers not only in transactional contexts, but also in non-transactional reliable and unreliable contexts. The road map towards the consolidation of the component-based infrastructure is shown in Figure 1. A general explanation follows:

**Componentization** – This phase implied the definition of functional dimensions characterizing persistence management. We tried to define fine-grain, but meaningful components, keeping in mind code reuse. In addition, we had to establish the boundaries of each data management aspect: persistence, fault tolerance, concurrent access, and transaction supports.

**Reasoning about dependencies** – We continued by establishing component inter-dependencies and frameworks for providing ways to enable components separation and/or assembling. In particular, we concentrated in characterizing complex relationships between persistence management and other system aspects, e.g. how to separate fault tolerance from persistence management?

**Customized persistence management** – Finally, we devised tools and techniques supporting multiple software configurations such as: persistence with no fault tolerance, persistence with crash recovery, and persistence with transaction support.

![Figure 1. Towards a component-based infrastructure: The road map](image)

The resulting infrastructure architecture is organized in three layers. The first layer copes with unreliable storage management. The second one adds support for system crash resilience, by introducing a redo-only recovery method. Finally, the third layer provides transaction support. All the layers are articulated with a multi-level recovery method.
1.2. Paper outline

The rest of this paper is organized as follows. The global picture of the architecture of our infrastructure is detailed in Section 2. This architecture is organized in three layers, and Sections 3 to 5 provide the design patterns characterizing the interactions between components of these layers. Then, Section 6 overviews our first prototype. Section 7 provides a short survey of related previous works on DBMS architectures and proposals for persistence service. We conclude and review ongoing work in Section 8.

2. Architecture

The infrastructure has a multi-layer architecture as shown in Figure 2.

![Diagram of Infrastructure Architecture]

Figure 2. Infrastructure architecture

Each layer in the infrastructure copes with a well identified set of system concerns. Layer 0, provides an unreliable persistence service. Layer 1 adds system crash resilience. Finally, Layer 2 adds support for implementing a transactional resource manager. A general description of components follows:

Layer 0 – Unreliable, base components

- The **CacheManager** is responsible of maintaining popular data objects in main memory. This component is based in the well known FIX/use/UNFIX protocol [GRA 93b]. This protocol allows to keep track of access on data objects to implement replacement strategies, and to maintain a reference count of concurrent accesses.
– The **StorageManager** provides an unified interface to access in an object-based fashion a given underlying storage product, e.g. a DBMS. Thus, this interface allows to plug-in not only *ad hoc* storage managers, but also any existing storage technology. For instance, storage managers can be built on top of object-relational DBMSs, by means of the appropriate mapping and runtime libraries.

– The **PersistenceManager** coordinates a **CacheManager** and a **StorageManager** and mediates object materialization, and update propagation. Note that this component does not include recovery-related code.

Layer 1 – System crash resilience

– The **LogManager** component provides a general purpose logging facility. The major responsibilities of this component are the implementation of reliable storage, log record organization, as well as housekeeping tasks such as log truncation. In order to keep generality, checkpointing and recovery methods are implemented elsewhere, e.g. in the **ReliablePersistMgr** and in the **TransPersistMgr**.

– The **ReliablePersistMgr**, meaning reliable persistence manager, adds system crash reliability to the base **PersistenceManager** component. In order to do that, this component uses a **LogManager** to implement a redo-only recovery method.

Layer 2 – Transaction support

– The **ConcurrencyControl** component provides an interface allowing to accommodate a large range of concurrency control methods.

– The **TransPersistMgr**, which stands for transactional persistence manager, allows to build transactional resource managers. Transaction support is based in a multi-level recovery approach described in Section 5. This component relies on a **ReliablePersistMgr** and a **LogManager** to implement transaction atomicity and durability, and on a **ConcurrencyControl** component for transaction isolation.

Some components can be used outside of the persistent object managers. This is the case of components **CacheManager**, **StorageManager**, **LogManager**, and **ConcurrencyControl**. The rest of components implement persistence management protocols.

The infrastructure can be used to built three families of persistent object managers. First, unreliable managers are built using only components in Layer 0. Second, reliable managers are built with components in layers 0 and 1. These managers provide a checkpointing-based programming model. Finally, transactional managers (e.g. XA compliant resource managers) can be built with components from the three layers.
3. Base components

Layer 0 is at the heart of our platform and provides components to assemble a family of unreliable persistent object managers. Using this base configuration supposes that system crashes do not impact negatively persistent data integrity. Examples of usage of such a configuration are web caching or read-only systems.

This layer is responsible of loading objects from and eventually writing them to underlying storage. There are three components in this layer, namely PersistenceManager, CacheManager and StorageManager. It is the PersistenceManager which takes the role of coordinator of the object loading and writing. Figure 3 provides more details about component interfaces.

Operations provided by PersistenceManager component \(^1\) capture hints on object accesses. Note that read and write operations are expanded to reflect temporal breadth of accesses. Thus, readIntention() and writeIntention() express the beginning of accesses on read or write mode, respectively. Further, once accesses on objects are finished (e.g. by transaction commit time), the readCompletion() and/or writeCompletion() should be declared. We assume, that there are not blind writes, that is every write is preceded by its corresponding read operation.

For ease of explanation we discuss sequences of operation calls in two parts. The first one corresponds to loading, i.e. object fault design pattern, and the second one to writing, i.e. update management.

---

1. Hereinafter, when we talk about operations provided by a component, we refer to operations defined in the corresponding interface. For instance, PersistenceManager provided operations are defined in the IPersistenceManager interface.
3.1. **Object faulting design pattern**

Object faulting is managed as it is done traditionally in DBMS. Object faulting due to prefetching and eager swizzling are out of the scope of our analysis. Figure 4 shows the design pattern for object faulting.

![Diagram of object faulting design pattern](image)

**Figure 4. Typical object faulting call sequence**

The sequence starts with a `lookup()` operation on `CacheManager` component. When the requested object is found, it is simply returned after being fixed in cache with `fix()` operation. This sequence can be derived from Figure 4 by eliminating operations enclosed within the dashed box. If the object is not currently present in cache, the `PersistentManager` will request a copy from storage, to the `StorageManager`. This is done by executing `load()` operation. Finally, the object is added to cache and fixed, by calling `addToCache()` and `fix()` operations.

3.2. **Update management**

Management of updates on objects consists of two parts. The first concerns providing means to identify updated objects that are present in the cache. The second issue concerns update propagation policies.

3.2.1. **Identifying updated objects**

Updated objects, also known as dirty objects, are usually managed within special data structures, such as hash tables or queues [MOH 92]. We have decided to manage such data structures by a sub-component called `DirtyObjectManager` (see Figure 5).

`DirtyObjectManager` provides, among others, the operation `markDirty()`. This operation is used to notify that a given object has been updated. Its usage is described in detail in the following subsection. Figure 5 shows `DirtyObjectManager` component and its interface, and Figure 6 shows how `markDirty()` operation is called.
3.2.2. Update propagation

In general terms, update propagation can be either immediate or deferred. Implementing an immediate update propagation scheme is straightforward. However, it usually incurs into high overheads, because it needs a large amount of write operations. A special case of immediate update propagation occurs in the transactional context, referred to as *force-at-commit* propagation [HÄR 83]. This latter is discussed in Section 5.

Concerning deferred update propagations, the problem is to determine where propagation decisions come from. This is in fact one of the most important sources of component interdependencies. Within Layer 0, only two types of events can be considered to manage update propagation, namely cache eviction and system shutdown events. The latter can be considered as a special case of cache eviction, where every cached object is evicted.

Figure 5 shows how update propagation policies are registered into the platform. A given DirtyObjectManager is associated to a set of update propagation event listeners. Whenever an update propagation event must be notified to the DirtyObjectManager, the corresponding event listener will catch it. The status of the object associated to the event is checked. If the object is dirty, its state is flushed to storage, with flush() operation. If the object is clean, its state can be simple discarded.

![Figure 5. Classes for update management](image)

Within unreliable persistent object managers, the only update propagation event listener is the CacheEvictionListener. Figure 6 shows how a cache eviction event triggers an update propagation.
3.2.3. More on update propagation hints

In addition to the cache eviction events, we can consider the following flushing hints.

**Checkpointing** – This is an activity that forces writing information to stable storage in order to reduce recovery time, and log space. Depending on the checkpointing algorithm, it may be required to flush some or even all updated objects that remain in cache, and some other system data structures.

**Transaction commit** – The *force-at-commit* policy is one possible strategy for update propagation management. It implies that every object modified by a transaction should be flushed at transaction commit time.

**Reconnection in mobile environments** – In mobile and in general in weakly connected environments, a reconnection event can trigger update propagation.

Some applications could explicitly provide hints to drive update propagation. Further, some advanced protocols such as the distributed group commit presented in [PAR 99] could be also considered.

Recently, Lomet and Tuttle [LOM 95, LOM 99] provided a quite complete formal framework to understand the impact of using logical logging on cache management, i.e. update propagation. As a result of this formalization, they proposed a set of algorithms to implement a general cache manager. These algorithms include one that calculates a write graph. This write graph is used by another algorithm that schedules cache writes. The latter can be included as an update propagation policy in our infrastructure.

4. Adding system crash resilience

This layer adds a first level of resilience to the platform. System crash resilience is related to the problem of bringing persistent data to a consistent state after a system crash, e.g. system persistence in non-transactional contexts, or transaction durability.
In our platform, system crash resilience is provided by the **ReliablePersistMgr** which implements a redo-only recovery mechanism. To this end, this component stores redo information within a log.

Figure 7 shows how components in Layer 1 extend those in Layer 0 to add system crash resilience.

![Figure 7](image)

**Figure 7. Components in Layers 1 and 0**

### 4.1. Collecting redo information

Supporting deferred update requires storing redo information, i.e. within the log. There are several formats for logging information. They are classified as physical, logical and physiological logging [GRA 93b]. The selection of the type of log information is left to the component implementor. Redo recovery information is gathered by the **ReliablePersistMgr** component when the `writeCompletion()` operation is requested, as shown in Figure 8. Once the corresponding log record is stored, the same `writeCompletion()` operation is delegated to Layer 0.

![Figure 8](image)

**Figure 8. Sequence diagram for acquiring redo recovery information**
4.2. Checkpointing

The primary goal of checkpointing is to limit the work to be done during recovery after a system crash. Checkpoints are produced periodically, so that all updates preceding a checkpoint are guaranteed to be stable. As said before, checkpointing might induce update propagations. We have classified checkpointing schemes according to the sequence of activities to be done. Thus, we distinguish the following types of checkpointing:

**Sharp or heavyweight** – These schemes require stopping all activities, flushing all updated objects currently in cache to underlying storage, and writing a end of checkpoint log record.

**Incremental** – In these methods, the dirty object table is managed in two or more parts, each one corresponding with checkpoints intervals. Thus, only dirty objects of the oldest checkpoint interval are to be flushed. Examples for this kind of scheme include the *penultimate checkpoint* described in [BER 87], and the Oracle incremental checkpoint [JOS 98].

**Lightweight** – By exploiting the undo and redo recovery information, this class of methods require storing only the system data structures, e.g. dirty object table, active transaction table. ARIES [MOH 92] supports this kind of checkpointing.

As suggested above, checkpointing may require storing the dirty object table in the log. The *serialize()* operation of *DirtyObjectManager* component is used to this end. The *deserialize()* operation is inverse to the previous one, and is used in the restart procedure.

*CheckpointListener* is added to the dirty object manager, in order to mediate update propagation induced by the checkpointing process. This is shown in Figure 9.

![Checkpointing listener](image)

**Figure 9.** Checkpointing listener in the context of update propagation policies

Figure 10 depicts a generic sequence diagram for the checkpointing process. Note that a subset of calls have numeric labels, which will be used in the explanation. As
a matter of fact, a sharp or heavyweight checkpoints incur in the execution of steps 2 and 4 on the sequence. The implementation of an incremental checkpoint requires the execution of steps 1, 2, and 4. Finally, a lightweight scheme needs the execution of steps 1, 3, and 4.

![Figure 10. Generic sequence diagram for checkpointing](image)

As Layer 1 does not deal with transactions, the sequence diagram shown in Figure 10 does not include a call for storing the active transaction table. However, storing the serialized form of the active transaction table should occur as part of step 3.

5. Adding transaction support

Layer 2 deals primarily with providing support to transaction abstraction. Hence, the issues to be addressed are transaction isolation, atomicity and durability. Atomicity and durability are related to recovery.

5.1. Transaction isolation

Transaction isolation is twofold. In the one hand, it implies providing a way to coordinate concurrent accesses. In the other hand, it concerns ensuring object consistency when the system maintains multiple versions of a single object. This is found when an optimistic concurrency control method is used, or in distributed contexts, e.g. client-side caching.

In our infrastructure, we have delegated the implementation for transaction isolation to the `ConcurrencyControl` component. Figure 11 shows the definition of this component.

The `IConcurrencyControl` interface has been derived from the visibility framework provided in [BLA 98]. The operations included in this interface enables the integration a large set of concurrency control, as well as cache consistency methods.
5.2. Transaction recovery

Atomicity and durability are ensured by underlying recovery methods, and they are commonly addressed together. However, we need to separate them to achieve componentization. To this end, we adopted a layered architecture approach, which is based on works about multi-level recovery [WEI 90, LOM 92, WEI 93].

As a layered system, durability is delegated to Layer 1, and atomicity to Layer 2. In order to do that, Layer 2 uses a high level compensation approach. That means that Layer 2 stores undo information, and when needed, such information is used to determine compensation operations.

Based in this framework, Layer 2 can add support to transactions. This consists in providing the operations related to transaction management, delegating isolation to the corresponding component, handling undo information and exploiting the durability guaranties provided by components in Layer 0 and 1. Figure 11 shows how TransPersistMgr, standing for transactional persistence manager, is related to the other components in the infrastructure.

Figure 11. Structure of components in all layers

Figure 12 shows the sequence of operations that characterizes transactional processing. Firstly, a beginTransaction() operation is called, prompting a beginTx log record to be written. For every object to be accessed, a writeIntention() should be called before any update can take place. This operation allows to record the undo recovery information. That corresponds with the actions taken at the Layer 2 to record

---

2. The sequence suggests that redo and undo information is stored by a single LogManager. With this assumption, the resulting management regime is similar to that of ARIES [MOH 92]. It is also possible to use two logs, but their management becomes complexer as noted in [WEI 90].
undo information. The redo information is handled by Layer 1, as described in Section 4. Note that actions on Layer 1 are accomplished when writeCompletion() operations are requested. In the sequence diagram, this is done at commit time, i.e. when commitTransaction() operation is called. Support for partial savepoints can be implemented in a similar way.

![Sequence diagram](image)

**Figure 12.** Typical call sequence for transactional access

During restart, after a system crash, redo information is sent to Layer 1, to be processed by the **ReliablePersistMgr** component. Similarly, undo information is sent to Layer 2, to the **TransactPersistMgr** component. The restart procedure is based in the repeat history paradigm proposed in ARIES [MOH 92], and generalized by the multi-level recovery methods [WEI 90, LOM 92, WEI 93]. The recovery method for our layered architecture is described in more detail in [Gar ].

Additionally, support for advanced transaction models can be provided. This can done, by implementing the algorithms proposed in [LOM 92], in the context of multi-level recovery.

### 5.3. Force-at-commit update propagation policy

As stated in Section 4, the transaction commit event can be taken as an update propagation policy. This corresponds to a force-at-commit cache management policy [HÄR 83]. However, implementing a force-at-commit policy implies that the underlying storage support an atomic propagation of any arbitrary set of objects.

Examples of atomic capable storage techniques include shadow paging, and intention lists [HÄR 83]. Note however, that **StorageManager** components implemented on top of full-fledged DBMSs can implement atomic propagation by using native transactions. This strategy is implemented in several commercial database middle-
ware products (see for instance [TIM 01]), and is used to guarantee maximal reliability of data. This is done by delegating fault-tolerance management to the underlying DBMS. The trade-offs between reliability and performance are obvious.

A force-at-commit strategy can be added to our infrastructure with the corresponding update propagation policy, e.g. **TxCommitListener**, to our **DirtyObjectManager**. This is shown in Figure 13. This component will force flushing all objects updated by a given transaction, by the time this transaction commits.

This base strategy can be extended by means of a group commit protocol. The idea is to group a set of transactions, and their corresponding updated objects, and request a single atomic update, i.e. native transaction. This can hopefully increase performance, by reducing the number of message exchanges between the infrastructure and the DBMS.

In the same way, in distributed contexts, the distributed group commit protocol described in [PAR 99], can be integrated by applying the same principles. These update propagation policies are shown in Figure 13.

**Figure 13. Some transaction related update propagation policies**

The pattern for update propagation policies was described in Section 3.

6. Implementation

We developed a first prototype of our infrastructure. We chose the Java language to exploit particularly the high level of portability of its bytecode. This is because we are interested in using our infrastructure in different configurations of hardware and operating system.

Based in the principles of architecture, we developed implementations for our components:

– Cache manager – The implementation has been done according to **CacheManager** interface and includes several replacement policies, namely LRU, MRU and FIFO.
– Storage manager – One implementation has been done on top of the Java provided file management libraries. Files are organized in pages, and objects are allocated inside slotted pages [CAR 86]. We have implemented a GiST [HEL 95] based index framework with support for B+ and R* trees. These indexes are used to maintain system information such as the mapping between logical object identifiers and in-disk physical addresses. Thus, we provided a complete object-based storage manager according to the StorageManager component interface. Note that we used the cache management components in the construction of the storage manager.

A second implementation has been done with a subsystem allowing to map Java objects into relational databases. To this end, we have used a open source software [OBJ ] which we have masked behind our StorageManager component interface.

– Log manager – As for storage management, the implementation of the log manager was done on top of Java provided file management libraries. It also uses the cache management facilities of our infrastructure. The implementation provides reliable stable storage as proposed in [GRA 93b].

– Concurrency control – A lock-based concurrency controller has been implemented as proposed in [GRA 93b]. Thus, this component implements a two-phase locking protocol. Deadlocks are resolved by timeouts.

– Unreliable persistence manager – We implemented the UnreliablePersistMgr component, along with the DirtyObjectManager component. This latter includes two update propagation policies, namely the CacheEvictionListener and CheckpointListener.

– Reliable persistence manager – This component has been implemented which uses a physical logging subsystem for storing after images. These images are used to implement the redo only recovery method for system crash resilience.

– Transactional persistence manager – We implemented a local transactional resource manager. This component implements the TransPersistMgr specification, and the multi-level recovery method sketched in Section 5. Hence, it is responsible of the undo-based recovery method for providing transaction recovery. Additionally, a primitive support for nested transactions is provided, because the lock-based concurrency controller does not support lock delegation.

6.1. A reliable non-transactional configuration

We tested a reliable non-transactional configuration of our infrastructure in the context of the PING project [PIN ]. This project aims at providing a platform for building large-scale massively multi-user software, such as large-scale simulation software, and networked games. We concluded that a transactional-oriented DBMS is ill-suited, justifying the use of our infrastructure. We are currently investigating further in this domain [NED 00].
6.2. A transactional configuration

We tested a transactional configuration for embedding it in a resource constrained environment, i.e. a handheld computer (Compaq iPAQ). This test is anticipating the development of a platform with support for mobile transactions as suggested in [SER 02].

Finally, we tested a transactional configuration on top a full-fledged relational DBMS. In this case, we have been able to add support to nested transactions. We will further test our infrastructure in the integration of persistence services for advanced EJB containers [JCP 00].

7. Related work

7.1. Extensible DBMS

By the end of 80’s, the industrial and academic research communities started efforts for developing what was called the extensible database management systems. In the following, we give a general classification of those approaches.

**DBMS generators** – The idea of generating some parts of the DBMS has been explored by some researchers. Examples of this approach include the Genesis project [BAT 88] and the Volcano query optimizer generator [GRA 93a]. This approach has been apparently abandoned, presumably because of the difficulties due to intrinsic complexity of the DBMS software.

**DBMS with an extensible type subsystem** – Some researchers studied ways to let users extend DBMS type system (i.e. ADT), and underlying access methods (i.e. indexes). Prototypes such as Postgres [STO 86] at the University of Berkeley, and Starburst [SCH 86] at IBM are examples of this approach. Note that this approach has effectively evolved, and it is at the origin of the so called object-relational DBMSs.

**DBMS kernel toolkits** – This approach has been taken in the Exodus [CAR 86], Shore [CAR 94] and DASDBS [SCH 90] projects. The idea was to build a minimal (core functionality) DBMS kernel along with a set of tools allowing to build ad hoc DBMSs. These projects have been testbed for lots of research developments. Note however that this prototypes were almost full-fledged DBMSs, so that lighter weight DBMS implementations could not be envisioned.

7.2. Persistence service standards

Among the proposals for persistence service standards, we can mention three of the most widely accepted: the Object Management Group’s Persistent State Service (PSS
2.0) [OMG 00], the Object Data Management Group standard (ODMG 3.0) [CAT 00], and the Java Data Objects (JDO) [JCP 01a] at the Java Community Process. All of them attempt to provide a standard approach to integrate existing persistence technology (e.g. DBMS) by using a single homogeneous set of interfaces. Effectively, they intend to provide a framework for building persistent applications independently of the underlying support, thus promoting in some scale multi-vendor interoperability.

The three proposals provide a transaction-based processing mode, except for PSS which also provides a non transactional approach. The proposals promote high level APIs (Application Programming Interfaces) hiding implementation details. JDO goes beyond revealing some system internals. Among those details, there are notably APIs to provide hints for cache replacement, hints for swizzling/object faulting scopes, for selection of concurrency control strategies (e.g. locking-based or optimistic approaches), etc.

The OMG has taken from the beginning an service oriented approach. Hence, they proposed the following persistent related services [OMG]: transactions, querying, and concurrency control. However, although PSS specifies interaction with the transaction service, nothing is said about interactions with querying and concurrency control services.

The Java Community Process has started defining a generic cache service [JCP 01b]. Note that the definition process is led by Oracle, so that a DBMS-like approach can be expected. However, there is no up-to-date information about interaction between JDO and the cited cache service.

7.3. Industrial ventures

OLE DB [BLA 96] is one of the industrial products that has explored externalizing DBMS-like services. OLE DB components can be classified as tabular data providers and service providers. Data providers are components that wrap data repositories in the form of relational tables. Thus, text files, spreadsheets, ISAMs, and even full-fledged DBMS can be exposed to OLE DB clients with as single, unified interface. Furthermore, a service provider is a OLE DB component that does not own data, but provides some service. Examples of OLE DB service provider components include query engines, and transaction monitors.

Another commercial product proposing an approach similar to ours is that of the Poet’s FastObjects j2 [SOF 01] (aka. Navajo). FastObjects j2 is in fact an embedded DBMS intended to be used in resource constrained mobile devices (e.g. PDAs). This DBMS has a highly modular architecture so that it is possibly to add/remove some components, i.e. XML import/export, logging and synchronization modules.
7.4. Our position

Earlier works have kept in mind the goal of building more and more complex DBMS. We argue, however, that the process inverse has to be done, trying to breaking down DBMSs in simpler but meaningful software components. Our vision is close to the RISC-style data management blocks described in [CHA 00]. Thus, instead of talking about an extensible DBMS kernel, we go beyond by proposing fine-grain software components for mechanisms such as caching, logging and recovery.

Our approach can be also compared to that of the JUPITER file manager inside Genesis [BAT 88]. JUPITER has been designed to accommodate several implementations for a same system concern. However, its design relies in the use of programming constructs available at the time the Genesis project was active, e.g. conditional compilation.

We have tracked the development of standards for persistence, and other related services. We have witnessed, unfortunately, the lack for integration between those services. We are convinced that the understanding of the subtle dependencies inside the DBMS machinery can be useful to define well integrated services. In addition, we claim that specifying services at lower level could allow to optimize resource sharing between services. For instance, a persistence service which specifies a caching sub-service, could allow different vendor implementations cohabiting at runtime to share a single instance of such sub-service. The same thing could be said for other low level services.

The OLE DB approach provides cursor-based navigation through tabular data. Hence, the central problem of that work was homogeneous access to data sources different to DBMS, and query services. Our work is complementary to that one, because we propose service components of lower level. We can envision to build OLE DB service components which use our caching, logging, concurrency control, and other components.

8. Conclusions and ongoing work

The continuous evolution of requirements in data intensive applications imposes new challenges to the research and industrial communities. DBMS architectures should be revisited, and we claim that a component-based approach should be taken.

We have presented the first results of our works looking at unbundling DBMS. Our efforts have resulted in the formulation of architecture principles, component definition, and characterization of component interactions. We focused on the isolation of persistence related services from the rest functions found in full-fledged DBMS. Based on our conceptual principles, we have completed an initial prototype. The usability of such prototype has been tested in several environments. It has been tested in networked gaming software, characterized by tight real-time constraints, and where transactions are ill-suited. We have carried out some preliminary tests in resource
constrained environments, and in database middleware. We also investigate how to integrate persistence services for EJB containers. Finally, we are developing benchmarks to compare our platform with existing monolithic software.

This work is only a single research thread of the NODS project. Thus, we work in the characterization of requirements of emerging application domains, such as, mobile systems, and large-scale computer supported collaborative work. In addition, we work hard on the analysis and characterization of other data management system concerns. They include replication management, fault tolerance, query processing, transaction coordination, and workflow management. Finally, we are interested in issues related to component assembling techniques.

9. Acknowledgements

We would like to thank all members of the NODS team, and particularly Tanguy Nedelec, Stéphane Drapeau, Claudia Roncancio, Elizabeth Pérez, Patricia Serrano, and Adnane Benaisa. Also, we thank Pascal Déchamboux and Alexandre Lefevre at France Telecom R&D.

10. References


[OBJ] ObjectRelationalBridge, “ObjectRelationalBridge project”,
http://sourceforge.net/projects/objectbridge/.


[PIN] Ping Team, “Platform for Interactive Networked Games.”,


[SOF 01] Software P., “FastObjects™ j2, White paper.”,


[TIM 01] TimesTen, “TimesTen Front-Tier 2.3: Dynamic data cache for Oracle 8i”,

