

# Transparent Access to Legacy Data in Java

Olivier Gruber

IBM Almaden Research Center

San Jose, CA 95120

## Abstract

We propose in this paper an extension to PJava in order to provide a transparent access to corporate data (files, relational systems, etc.). This extension relies on the concept of external object faulter which is conceptually close to the external pager one in operating systems. Our approach is a good basis for building middleware. Our prototype shows the feasibility of our approach.

## 1 Introduction

Accessing corporate data is a more and more crucial issue for any desktop environment, Java is no exception. Java currently offers the Java DataBase Connectivity (JDBC) framework which aims at playing in the Java realm the same role that ODBC played in the PC world. However, JDBC does not provide transparent access to corporate data. It is interesting to realize that such a transparent access is complementary as well as contradictory with persistence in programming languages.

Persistence in programming language is concerned by making some created objects to persist accross program executions. Criteria to decide which objects have to persist differ depending upon the adopted persistence model (reachability, inheritance, persistent allocation). However, the universal issue is how to keep on stable storage some objects, created during earlier program executions, in order to re-access them at a later time.

Transparent access to corporate data targets the converse problem of accessing existing legacy data and mapping them into a programming language. Providing such a transparent access has the same advantages than providing transparent persistence. The basic idea is to be able to view legacy data as collections of objects. For instance, relations of tuples or files of records become collections of objects. These collections can be browsed but also queried if one is to provide a query processor and a query engine in the language.

Transparent access to corporate data can be looked at as an extension to a persistent language. In this paper, we will assume the existence of a persistence

framework such as the one in PJava [2] which provides both persistence by reachability and a transactional framework. Our approach is based on the notion of an external object faultler, conceptually very close to the concept of external pagers in operating systems. The external object faultler is in charged of fetching or putting away object states, being driven by the PJava run-time.

The mapping between object classes and the legacy systems is under the responsibility of each external object faultler which wraps a legacy system. For example, a relation of tuples in a relational database describing employees could be mapped onto a collection of employee objects. The external object faultler would be responsible for mapping tuple attributes onto class attributes, foreign key attributes onto object references, etc.

This paper is structured as follows. In section 2, we sketch a typical partition architecture for a persistent language which allows for an external object faulting mechanism. Section 3 describes in detail the protocol between an external object faultler and the PJava runtime. In Section 4, we present our current validation prototype. Our implementation is totally in Java and do not rely on PJava which is simulated. In Section 5, we conclude.

## 2 A Partitioned Approach

An external object faultler (XOF) is a similar concept to the external pager one in operating systems, except that XOFs apply to objects and not pages. We decided to look at XOFs as an extension to a persistent language. It may be possible to provide XOFs for non persistent languages, however we feel having a persistent host language simplifies the design and provides potentially more power, especially when one considers transactional persistent languages such as PJava [2].

The key point addressed by external object faultlers is that some objects persist outside the standard persistence mechanism of the language. This has two consequences. First, the language run-time cannot be responsible for fetching and saving all object states. Second, it is not solely responsible any more for the persistent identity of objects. We believe in an open approach where the language run-time delegates parts of the persistence task to some externally defined agents (XOFs) while remaining the overall coordinator.

The key idea is to partition the object store beneath the Java virtual machine, each partition being backed up by a different XOF or by the native persistent manager. In such a context, PJava plays the role of a coordinator. Stabilize calls as well as transaction commits become distributed two-phase protocols between XOFs.

Not only that, but the notion of identity has to take into account the different partitions, that is, dealing with local and cross-partition references. Each XOF provides persistent identity in a partition-local manner (LID). However, objects may reference each other across partition boundaries. This requires

PJava runtime to provide OIDs which are system-wide object identifiers. Like in most partitioned or distributed systems, a system-wide OID is composed of the identifier of the partition and the local identifier. This is a well-known mechanism. PJava maintains the cross-partition reference tables.

Notice that LIDs are XOF-dependent, that is, the content and structure of a LID depends of the legacy sources which is wrapped. LIDs may take many forms from simple number (offset in files) to multi-valued primary keys in relational database systems. Also notice that some legacy system may not be able to accept OIDs in place of LIDs. Relational systems are a good example if this case. The problems appears if one assumes that legacy applications will still run on top of the relational database or that SQL will still be used from PJava to benefit from the power of queries. In both cases, relational foreign keys cannot be replaced with OIDs: neither legacy applications nor SQL would be able to handle these.

### 3 External Object Faulter

The role of the external object faulter is to fetch or flush object states on behalf of the PJava runtime. It is interesting to detail the object faulting mechanism.

#### 3.1 Object Faulting

One of the main issue here is to determine which mechanism should be provided to write XOFs. On the one hand, XOFs can be thought of as plug-ins. In this case, XOFs are trusted software modules (like the current JDBC-ODBC bridge is) and raw C pointers are used to hydrate object shells. This approach provides maximum performance but very limited portability. On the other hand, one may wish to have XOFs implemented in Java. Indeed, XOFs for mapping files or JDBC sources would be great if they could be downloaded as applets.

In this later case, it is important that security be not compromised by XOFs. Therefore, care must be taken in the way XOFs are allowed to fill in empty object shells. Our idea is to introduce an interface XObject. XObject provides generic methods to set and get attributes. These methods access attributes on a name basis, that is, the class name and the attribute name<sup>1</sup>. Note that a complete type orthogonality is achieved since classes do not have to be defined as supporting the XObject interface nor they have to implement the get/set methods.

XObject interface is in fact how an XOF sees an object shell, not through the object real type. The behavior of the get/set methods are obvious for basic types such as int or char. The string and array cases are more delicate. Although strings and arrays are Java objects, most legacy systems treat them

---

<sup>1</sup>Having both a class and an attribute names is necessary because attribute shadowing is allowed in Java between a subclass and a super class.

as values. For instance, a relational database cannot provide an identity for strings, only tuples have an identity. Therefore, the get/set methods for string or array attributes will have two flavors: a value flavor and an object flavor. In the value flavor, the get/set methods are simply taking a Java string/array as parameter.

The case of the object flavor belongs to the general case of object attribute. What should be the behavior of get/set methods when the attribute is an object reference? Well, this is where swizzling and unswizzling occurs. Everything starts by binding a root (a named object). The PJava runtime keeps track of the association (name,XOF). When an XOF binds a root, it will ask for a creation of an object from the PJava runtime, providing the oid of that object, it will get an XObject back. Using that XObject interface, the XOF will be able to set the attribute values. For object attributes, the XOF must provide the OID of the referenced object. The PJava runtime will remember the OID and swizzle it — checking for already resident objects (thereby avoiding duplicates). Upon flushing, the XOF will get back OIDs when asking for the value of object attributes.

In some sense, the XOF never sees the Java references to the object it loads. It only sees XObject and OIDs. Notice the above scheme simplifies XOF implementations. Indeed, all the burden of providing a fullfledge object manager is under PJava responsibility. This requires PJava to maintain the object cache, to keep track of resident objects (avoiding duplicates) and of updates. Also, it is responsible for swizzling/unswizzling object references.

## 3.2 Schema Mapping

An important issue in wrapping legacy sources is the mapping technology used. Both upon object faults and flushes, XOFs have to know the type information of objects as well as the necessary information for mapping the object states onto their legacy system.

One could state this is the XOF implementor problem. The XOF has then to keep schema information on objects. This means that given an OID<sup>2</sup>, an XOF has to be able to find the corresponding schema description. If this works fine, this induces much redundant work. Indeed, the PJava runtime already keeps a detailed schema about classes and is able to relate objects with their class description. Allowing the XOF to access that information would really simplify the XOF implementation.

However, this is not enough. Most of the time, a mapping is necessary. For instance, a name mapping may be needed between the column names in a relational database and the attribute names in the object. Similarly, the table name needs to be mapped to the class name. Therefore, XOFs need to associate a class definition with mapping information. Again, if PJava would keep that

---

<sup>2</sup>Provided along with the XObject interface

information, the implementation of XOFs would be even more simplified. In other words, PJava would allow XOFs to attach mapping information to type definitions, mapping information provided along with the XObject interface upon object faults and flushes. This way, XOFs leverage to its full potential the fact that PJava has to manage a persistent schema.

## 4 Implementation

Our current implementation differs from the above design, mainly because PJava is not yet available and because we lacked man power to modify the Java interpreter. However, the philosophy is respected. Our prototype is entirely written in Java and composed of a schema and two XOFs, one for files of records and another for relational database systems. The XOF for relational database systems is underconstruction, it uses JDBC to access relational sources.

Having a schema is necessary to keep track of object types, persistent roots and collections, as well as the mapping from legacy data sources to object classes. Our schema is based on Java type system and is totally implemented in Java. We developed a class loader in Java which is able to read class file format. We thereby have a fullfledge run-time description of Java classes, within Java. The mapping framework is very primitive, mostly supporting name mapping and foreign key translation into object references.

Our prototype essentially diverges from the above design around the object swizzling mechanism. The only possible strategy for swizzling in a safe language like Java seems to be an half-eager half-lazy approach. In a eager approach, when an object is hydrated (loaded with its state), all the references it contains are swizzled, that is, the pointed-to objects are created and their state loaded in memory. In a lazy approach, the pointed-to objects are neither created nor their state loaded. Without support from the VM, none of the above schemes is possible. However, an intermediate approach works. When hydrating an object, the responsible XOF will create the pointed-to objects, but will not hydrate them (their state will be not be loaded). By creating the pointed-to objects, the XOF gets the Java references it needs for the hydrate process.

In other words, if an object A references an object B, when A is hydrated, the object B is created as a side effect, but as an empty shell. That way, the XOF obtains a Java reference for B for swizzling the reference in A pointing to B. Given that we create an empty shell, we avoid the recursive swizzling of an eager strategy.

Of course, some special course of actions has to be taken to fetch states of empty object shells before they are actually used. Objects which are managed by XOFs having to be instances of classes deriving from a special persistent class (PObject). So long for transparency, but again, this is due to the absence of collaboration from the Java VM. The PObject class provides a method called hydrate() which needs to be called before an object state is accessed. Also, the

PObject class provides a method `mark_dirty()` which signals updates.

## 5 Conclusion

We proposed in this paper an extension to PJava in order to provide a transparent access to corporate data. This extension relies on the concept of external object faulter. The necessary protocols between PJava runtime and external object faulter are simple. Security is not violated, allowing external object faulters to be written in Java and therefore being safely downloadable.

External object faulters can be used to wrap legacy systems, thereby providing a seamless access to corporate data which reside in file systems or relational database systems. This is an extremely interesting cornerstone for building middlewares such as Garlic [1]. Middleware of this categories target seamless access but want to retain efficient query capabilities across multiple legacy systems. The challenge is to support queries while retaining an impedance mismatch as minimal as possible.

Our prototype shows the feasibility of our approach, even though PJava functionalities were simulated. We also intend to pursue in the middleware direction, implementing a query processor and a query engine in Java.

The biggest challenges of our approach are undoubtedly the transactional issues which are unavoidable when one talks about legacy systems. The impedance mismatch today is maximal regarding transactions. Transactional support varies greatly in legacy systems, from nothing to mostly short flat two-phase locking transactions. This is far from enough for most applications which end developing a lot of difficult and error-prone code. An airline booking system demonstrates the short comings of current transactional supports and the amount of coding effort which is needed. Providing the right transactional framework is a key issue in the success of PJava for the market segments of middlewares and business objects.

**Acknowledgement** I would like to thanks the people in the Garlic group for introducing me to middleware problems and architectures. Our discussion undoubtedly seeded the ideas presented in this paper.

## References

- [1] M. Carey *et al.* Towards heterogeneous multimedia information systems: The garlic approach. In *Proceedings of IEEE RIDE Workshop*, 1995.
- [2] M.P. Atkinson, M.J. Jordan, L. Daynès and S. Spence. Design issues for persistent java: a type-safe, object-oriented, orthogonally persistent system. In *Seventh International Workshop on Persistent Object Systems*, 1996.