Chapter 8

Persistence

This chapter covers the management of distributed persistent data, i.e. data that have a process-independent lifetime. A persistence middleware system mediates data transfer between persistent instances, residing on a data store, and their images in memory. We discuss the main issues related to the design and implementation of such a system. The chapter concludes with a presentation of JORM, a generic framework for persistence management, and a description of its use for developing persistence middleware systems.

8.1 Introducing Persistence

Persistence is an attribute of data that qualifies its lifetime with respect to that of the processes which operate on it. A piece of data is said to be *persistent* when its lifetime is independent of that of its creator process. Typical examples of persistent data are the contents of files or databases, as opposed to variables in common programming languages. The lifetime of a variable is at most that of the process which executes the program (it may be shorter, e.g. for a local variable declared in a procedure), whereas a file or a database item persists until explicitly removed.

Persistence is usually achieved by storing data on disks, which guarantees both persistence and permanence[1].

A persistent piece of data therefore exists in two forms: a memory instance, e.g. a Java object; and a data store instance, e.g. a record in a file or a row in a relational DBMS. A mapping must be maintained between these two forms, in order to (a) allow reads or writes on the memory instance to be interpreted by data transfers from or to the data store; and (b) translate references between memory instances into references between the corresponding persistent data store instances, and vice versa (this translation is known as *swizzling/unswizzling*).

[1]The notion of persistence is distinct, in principle, from that of permanence (or availability). Data are permanent if they remain available in spite of failures. The means of achieving permanence depend on the class of failures from which immunity is required (see Chapter 11 for details). Storing data on permanent storage (e.g. disk) ensures availability in the face of power failures. N-replicated copies on independent disks resist N-1 disk failures. Permanent data are persistent as well, but the converse needs not be true. For instance, persistent data could well reside in main storage; usually, however, persistent data are stored on disk to ensure permanence as well.

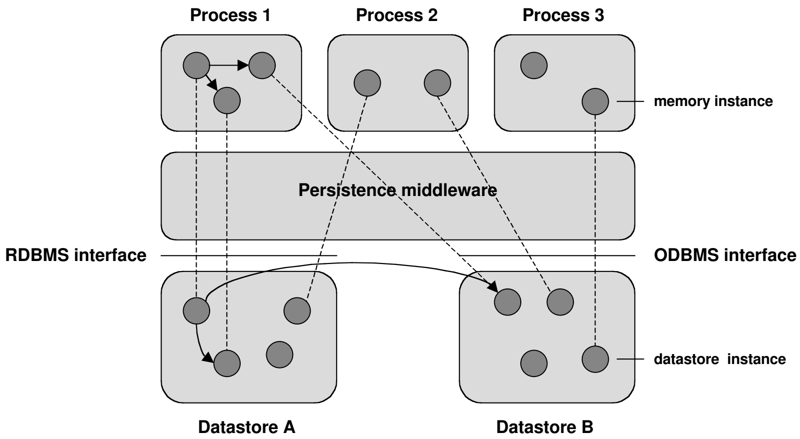

Figure 8.1. Persistence middleware

The function of a middleware for persistence support (Figure 8.1) is to manage that mapping. To enhance reusability, the provided APIs should be as generic as possible, i.e. they should not be tied to a specific data model or to a specific data store. Several such systems have been specified and implemented, including the CORBA Persistent State Service, the EJB Container Managed Persistence (CMP) and Java Data Objects (JDO).

8.2 Issues in Persistent Data Management

*** to be provided:

a discussion about the notion of persistence being common to OS/DB/PL, and the various approaches.

8.2.1 Specifying Persistent Objects

When trying to integrate database and programming language notions, the first issue is to decide which data should be persistent. While a number of ad hoc solutions have been proposed and experimented with, a general principle has been formulated under the name of *orthogonal persistence* [Atkinson et al. 1982]. It involves the three following properties [Atkinson and Morrison 1995]:

- Persistence independence: the form of a program is independent of the persistence of the data that it manipulates[2].
- Data type orthogonality. All data objects should be allowed the full range of persistence irrespective of their type.
- Persistence identification. The mechanism for identifying persistent objects is not related to the type system.

[2]This property is the analog of "access transparency" for distributed data.

Orthogonal persistence is a desirable property because it frees the programmer from concerns unrelated to the logic of the application and because it reduces the amount of code to be written. However, it has been found difficult to achieve in practice without compromising performance. In spite of the strong arguments in its favor, orthogonal persistence is not widely implemented (a recent example is PJama, an orthogonally persistent Java [Atkinson 2000]).

A type-independent way of defining persistence is by reachability: a data object should persist as long as it is accessible by a chain of references starting from a predefined set of persistent roots. Conversely, allocated storage whose contents has become unreachable should be freed by a garbage collector. Persistence by reachability is the most common way of implementing orthogonal persistence.

*** to be provided

how to specify persistent data: language, descriptors, etc.

8.2.2 Managing Data Transfer

When to load/store an object - explicit vs implicit

8.2.3 Naming Issues

*** to be provided

*** Naming an object - recall standard stuff

* persistent "universal", secondary storage aware name

* local name (address)

swizzling is the mapping between the two above names.

name resolution time: eager vs lazy, examples

one-level store, single address space

8.2.4 Data Stores

*** to be provided

*** specific discussion on object/RDBMS mapping

8.2.5 Implementation Issues

*** to be provided

8.3 Middleware for Persistence

*** to be provided

*** brief descriptions of PSS, CMP, JDO (principles and implem. issues)

*** common patterns for persistence, based on the above ("personalities" on top of core)

8.4 Case Study: The JORM Persistence Framework

JORM (Java Object Repository Mapping) is an original attempt at providing a common framework for building a persistence service, using a minimal set of generic concepts and tools based on the `export-bind` pattern (3.3.3). Several such services, or "personalities", have been built on top of JORM, including an implementation of EJB CMP (part of the JOnAS EJB system) and an implementation of the JDO specification. In the rest of this section, we present the main concepts and the architecture of JORM, and its use in implementing the two above personalities.

8.4.1 JORM Concepts and Architecture

JORM maps memory instances (MIs) on persistent objects (Data Store Instances, or DSIs). MIs are Java objects. The structure of a DSI is defined by the specific organization, or Data Store (DS), to which it belongs. Examples of DSs are a relational database (RDBMS), an object-oriented database (OODBMS), or an LDAP directory.

JORM is independent of any specific persistence model, i.e. the definition of which objects are persistent is outside its scope and must be provided by upper layers. Therefore JORM supports orthogonal persistence, but also allows type-specific persistence models. JORM defines persistent classes, i.e. classes whose instances are persistent, in the form of tuples and collections.

Names and Naming Contexts

Persistent objects are designated by instances of `PName`, which are "persistence-aware" names i.e. they include information (e.g. a primary key in a RDBMS) allowing the named object to be located on a data store (although that information is hidden in the implementation of `PName` and is not directly accessible to its users). `PName`s are defined in `PNamingContexts`, which are used for two main purposes.

- Federating several data stores, by defining naming contexts that span the stores to be federated. Thus an object located on a store can refer to an object located on a different store, possibly using a different organization.
- Allowing persistent objects to include polymorphic references, i.e. references that may designate objects of different classes. The context of the `PName` of a polymorphic reference must include names for all the contexts (more precisely, binders, see 8.4.1) corresponding to the possible classes to which the reference may belong.

The operations defined in Chapter 3 (`export`, `resolve`, `encode`, `decode`) apply to `PNames` and their contexts. A persistent object may be designated by a chain of `PNames` across several naming contexts, in which case the recursive resolution scheme described in Chapter 3 applies.

Bindings, Class Mappings, and Binders

The main abstraction of JORM is a *binding*, an object of type `PBinding` that mediates data transfer between a MI and a DSI. A JORM binding is the analog of a session in a communication protocol (Chapter 4). JORM bindings have been designed to accommodate a wide variety of uses and need to be specialized for several aspects.

1. *Persistent object structure and storage organization.* Bindings of a given persistent class are created by a factory called a *class mapping*, of type `PClassMapping`, which is specific to that class and to a particular data store organization.

2. *Naming scheme.* Bindings use `PNames` (8.4.1) to access data store instances. A binding is enabled (i.e. can actually give access to a DSI) if it is associated with the `PName` of the DSI. This is done by a binder, of type `PBinder`, associated with the class mapping (more details in 8.4.1).

3. *I/O operations* A binding provides generic read and write operations, which are specialized for the MI's class through an *accessor* that is passed as a parameter to each read or write operation (Figure 8.2). An accessor defines "getter" and "setter" methods for the different fields of the MI. These methods are specific to each class, which provides its own implementation of the accessor by specializing a generic `PAccessor` interface.

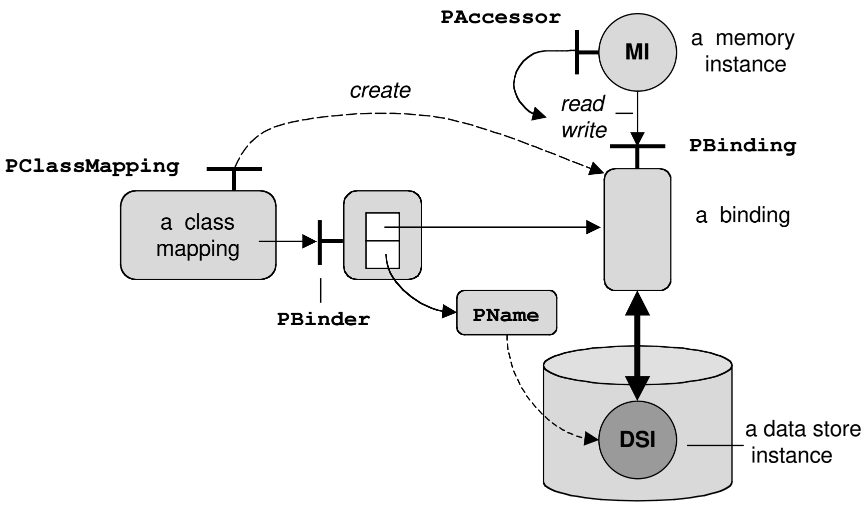

Figure 8.2. Binding a memory instance to a data store instance

Data store independence is achieved by using mappers. A *mapper* is an abstraction of a particular data store (e.g. a relational data base, an object data base, etc.). It acts as a registry for binders of the persistent classes that reside on the store. Several mappers may coexist, e.g. one for each different data store.

The classes that implement accessors, binders and class mappings are generated from description files: a file for each persistent class, describing the abstract structure of the objects, both in memory (to generate `PAccessor`) and in storage (to generate `PClassMapping`); the file includes a section for each mapper, describing the properties of the associated data store.

The Framework in Operation

We illustrate the internal working of the JORM framework through three typical operations of an object's life cycle.

a) *Binding to a persistent object.* The initial step of binding a MI to a persistent object is to create a (non enabled) binding object (call it `link`), by the operation:

```
link = mapping.createPBinding()
```

where `mapping` is the class mapping associated with the object's class. Two cases may occur.

1. There is no DSI (e.g. because the MI is new and has never been saved). Then a placeholder must be created for the DSI, through the following operation:

   ```
   the_name = link.export(connection, hints)
   ```

   where `connection` is a reference to the data store, and `hints` contains optional additional preferences. The operation creates and returns a new `PName`, which designates the new DSI. Note that the DSI is not initialized; this has to be explicitly done by a `link.write()` operation.

2. The DSI exists. Let `the_name` be its `PName`. Then the binding is enabled by the following operation:

   ```
   link.bind(the_name)
   ```

   which causes the binder to create the association between the binding and the name of the persistent object.

Both `export` and `bind` are delegated to the binder associated with the target binding object.

b) *Writing a persistent object.* A `write` operation to a persistent object through its `PBinding` interface copies the MI to the DSI. Two cases may occur when writing a field.

- The field contains (typed) data. A type-specific transfer procedure applies; it is determined by the `PAccessor` associated with the object.

- The field contains a reference to another persistent object, which has the form `field_id`: `ref_name`, where `field_id` identifies the field and `ref_name` is a `PName`, which may refer to a different naming context, or even a different DS. The following operations are performed:

  ```
  ref_name = get_field_id()                   (1)
  local_name = NC(field_id).export(ref_name)  (2)
  save(NC(field_id).encode(local_name))       (3)
  ```

In (1), the reference is read, using the specific "getter" for `field_id`. In (2), the reference (a `PName`) is exported to the local context associated with the field (these contexts are maintained by the class mapping). In (3), the name is encoded, using the local encoding conventions, and written to persistent storage. In effect, this sequence of operations performs swizzling. This method is general and applies to any form of naming through a chain of naming contexts.

c) *Reading a persistent object.* We only consider the unswizzling problem, i.e. reading a field that contains a reference to a persistent object. The operation sequence is the reverse of that used for writing:

```
local_name = NC(field_id).decode(read_stored_name) (1')
ref_name = NC(field_id).resolve(local_name)        (2')
set_field_id(ref_name)                             (3')
```

In (1'), the stored reference is decoded in the local context. In (2'), the name is resolved, possibly traversing a context chain, up to its original context. In (3'), the name is written to the MI, using the adequate "setter" method.

8.4.2 Using the Persistence Framework

Implementing Personalities

The main role of a JORM personality is to implement a MI manager. Such a manager defines the memory structure (implementing the `PAccessor` interface) that hosts the persistent data to be loaded or stored from or to the underlying data store. It also defines the way bindings are created, enabled and deleted, as well as the I/O synchronization points (i.e. at which time objects are read or written). We illustrate these principles through two personalities that have been built using JORM.

a) *The EJB/CMP personality.* The EJB specifications define the CMP[3] profile for persistent objects called *entity beans*. The associated programming model hides most of the complexity of persistence management in a *container*, whose role is to transparently support technical constraints that are specified separately from the code of the bean. The persistent fields of a bean are accessed using getter/setter methods. A reference to another bean is bound at run time to a particular bean implementation, whose class is uniquely determined (i.e. there is no need to support polymorphism). Since all field accesses are interpreted, lazy swizzling is used to set up the binding between beans by means of container interception objects, one of which is a JORM binding for managing I/Os.

EJB/CMP defines a single naming schema: a bean is identified by a persistent name, the primary key, which is composed of a subset of its persistent fields. There are three instances in which `PName`s are involved with beans:

- at creation time: the container creates and exports a `PBinding`, thus creating the associated `PName` using the relevant field values.

[3] Container Managed Persistence

- at look up time: the look up is performed by the `findByPrimaryKey` home method that requires a unique primary key object, a particular form of a `PName`.
- at traversal time: the `PName` that represents the reference is bound at first traversal, i.e. it is associated with a binding (which may itself be created if it does not exist), as explained in 8.4.1.

Using the `export-bind` pattern allows easy evolution of the personality without any change to its design. For instance, if the EJB specifications were extended to support polymorphism, one only would have to change the naming contexts associated with references to deal with `PNames` belonging to the binders of all bean classes that conform to the expected interface. Another example would be allowing a reference to designate beans of the same class stored in different, possibly heterogeneous, data stores. This only entails changing the naming contexts associated with references.

b) *The JDO personality.* The JDO specification aims at providing transparent persistence to almost any kind of Java objects[4]. This is achieved merely by enhancing the byte code of classes, thus making them "persistence capable". The added byte code has two functions:

1. intercepting all accesses to persistent fields, replacing them by a call to an associated getter or setter method (like in EJB, but transparently for the programmer).
2. providing a mechanism to load and store these persistent fields, which means integrating the functions of `PBinding` and `PAccessor`, and managing the whole life-cycle of a JDO instance (i.e. an object of a "persistence capable" class).

JDO may use two naming schemes: primary keys, like EJB/CMP; and system-managed names, under the control of the JDO implementation. Both support polymorphic references. The latter scheme has been implemented, by providing the relevant naming contexts for reference fields. References are swizzled at first traversal, through binding to a JDO instance, which may itself be created if needed.

Like EJB/CMP, this scheme may be easily extended to accommodate new requirements such as supporting references to other data stores or to remote objects (`export`ing the reference would get the distributed name from its stub, while `resolv`ing it would recreate that stub).

These experiments show that the JORM abstractions allow a quick implementation of persistence support, with a wide range of design choices. The flexibility of the `export-bind` pattern leaves room for easy further evolution.

Experience

A number of benefits have been derived from using the `export-bind` pattern in JORM.

As regards naming:

- The naming framework for inter-object references is orthogonal to the data model, which contributes to separation of concerns and allows independent evolution of the data model.

[4] JDO do not in a strict sense implement orthogonal persistence.

- Using contexts for naming references provides flexibility, by allowing any number of indirection levels to be transparently added to accommodate changes in the organization of the data stores.
- The `encode-decode` functions allow the names of persistent references (`PNames`) to be easily adapted to any specific format imposed by the data store (e.g. when integrating a legacy database).

As regards binding:

- Using mappers as a naming context for binders again helps to separate concerns, by isolating the data store specific information from the logical organization of the data.
- The notion of a `PBinding` allows for a clean definition of the persistent objects' life cycle by separating the preparation and enabling phases of a binding (respectively implemented by `export` and `bind`).
- By extending `PBinding`, or by inserting additional filters in the binding chain, the system may conveniently be extended and adapted (e.g. by adding encryption, compression, etc.).

8.5 Case Study: the Medor Query Framework

*** to be provided (maybe...) a section on queries illustrated by Medor

8.6 Historical Note

The designers of early programming languages did not pay much attention to persistence. The only way to make a variable persistent was to write it explicitly to a file and to retrieve it subsequently; there was no simple means of keeping track of the name of the file between different program executions. In addition, file operations in programming languages depended on the local operating system, and were not usually included in the standard language definitions.

In the early 80s, the notion of persistent programming (i.e. programming with implicitly persistent data) was introduced [Atkinson et al. 1983]. Several experimental languages for persistent programming were defined, usually as extensions of existing languages. In the same period, database programming languages were created in order to better integrate the operations on a database (queries, insertions, etc.) and the processing of the data. A key issue in both classes of languages was the relationship between the type of a data and its persistence properties. The principle of orthogonal persistence was formulated, but was not widely applied. An extensive survey of early work on persistence and data types may be found in [Atkinson and Buneman 1987].

In the mid-80s, object database management systems (ODBMS) were introduced, in an attempt to bridge the gap (often referred to as "impedance mismatch") between data processing based on object-oriented languages and request processing based on a specialized query language. An ODBMS is based on an object data model, including the notions of classes, types, inheritance, and method overloading. Data is stored in the form of persistent objects, and can be directly manipulated through an object-oriented programming language (usually an extension of an existing language). Objects are linked together through references. By the early 90s, several ODBMS were in existence, both as research prototypes and commercial products: Exodus [Carey et al. 1990], ObjectStore [Lamb et al. 1991], O_2 [?]. The notion of an object store was introduced in the same period, extending the concept of virtual memory to objects. An object store hides the physical management of the data from its users by providing the abstraction of a potentially unbounded one-level persistent data repository. An object store may be used as the lower layer of an ODBMS, or as the storage support of a persistent programming language. Examples of object stores are Mneme [Moss 1990] and Texas [Singhal et al. 1992].

In the late 90s, the fast development of the Internet and the World Wide Web promoted new frameworks for applications, based on interoperability standards such as CORBA and EJB and new data representation standards such as XML. ODBMS progressed at a slower rate than expected and remained confined to niche markets. However, the technology developed in object stores and ODBMS is now reused to provide persistence support for the products based on the new interoperability standards (e.g. CORBA Persistent State Service, EJB persistence management, Java Data Objects). Since the vast majority of legacy database systems are relational, smoothly interfacing objects with relational databases is an objective of the new persistence management systems. In the long run, it may be expected that the entire World Wide Web will become a vast distributed database system, with challenging perspectives for managing persistence on a really large scale.

References

[Atkinson et al. 1982] Atkinson, M., Chisholm, K., and Cockshott, P. (1982). PS-Algol: an Algol with a Persistent Heap. *SIGPLAN Notices*, 17(7):24–31.

[Atkinson 2000] Atkinson, M. P. (2000). Persistence and Java - A Balancing Act. In *Proceedings of Objects and Databases, International Symposium at ECOOP 2000*, volume 1944 of *Lecture Notes in Computer Science*, pages 1–31, Dittrich, K. R. et al., Eds. Springer.

[Atkinson et al. 1983] Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P., and Morrison, R. (1983). An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365.

[Atkinson and Buneman 1987] Atkinson, M. P. and Buneman, O. P. (1987). Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2):105 – 190.

[Atkinson and Morrison 1995] Atkinson, M. P. and Morrison, R. (1995). Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3):319–401.

[Carey et al. 1990] Carey, M. J., DeWitt, D. J., Graefe, G., Haight, D. M., Richardson, J. E., Schuh, D. T., Shekita, E. J., and Vandenberg, S. (1990). The Exodus Extensible DBMS Project: An Overview. In D. Maier and S. Zdonik, editor, *Readings on Object-Oriented Database Systems*. Morgan Kaufmann, San Mateo, California.

[Lamb et al. 1991] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. (1991). The objectstore database system. *Communications of the ACM*, 34(10):50–63.

[Moss 1990] Moss, J. E. B. (1990). Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems*, 8(2):103–139.

[Singhal et al. 1992] Singhal, V., Kakkad, S. V., and Wilson, P. R. (1992). Texas: An Efficient, Portable Persistent Store. In Albano, A. and Morrison, R., editors, *Proceedings of the 5th International Workshop on Persistent Object Systems,* San Miniato, pages 11–33. Springer.