Chapter 6: Proof-of-Concept Implementation

In Chapter 4, we motivated the need for enterprise applications with custom and adaptable process topologies. Then, in Chapter 5, we presented our FPT architecture, which consists of a set of concepts for realizing enterprise application middleware that supports custom and adaptable process topologies. In this chapter, we outline the design and implementation of our *Free Data Objects* (FDO) framework, which is a proof-of-concept implementation of an enterprise application middleware that is based on the FPT architecture.

Our FDO framework provides common infrastructure functionality for enterprise applications that follow the "multi-tiered enterprise application" architectural style defined in Chapter 3. The framework is data-centric – its main task is to provide enterprise applications with efficient, distributed, transactional access to business data objects that are persistently stored in one or more data stores. All concepts of the FPT architecture have been implemented – in particular, this means that a software architect can realize arbitrary custom process topologies through configuration. Also, an architect can easily adapt process topologies of existing enterprise applications through reconfiguration, which is important when requirements change. All process topology patterns described in Section 3.4 are supported.

The FPT architecture focuses on fundamental, high-level design decisions – many medium and lower-level details are left to implementations and, thus, there is still considerable freedom for realizing FPT-based middleware implementations. The purpose of this chapter is to present our particular implementation as an example of how the concepts of the FPT architecture can be realized and fit together in a concrete system. An evaluation of the FPT architecture and the FDO framework can be found in the following Chapter 7.

A detailed description of all design and implementation aspects of our FDO framework is clearly out of scope. Instead, we focus on

- the top-level design of our framework,
- the design and implementation of concepts of the FPT architecture,
- core data structures, and
- design decisions that have a major impact on performance.

The latter are important because they have a heavy impact on the results of the evaluation presented in the next chapter.

This chapter is structured as follows: In Section 6.1, we give an overview of the FDO framework and describe its top-level architecture. Section 6.2 shows how application developers can define process topology and data distribution scheme through configuration. Section 6.3 describes how communication between object managers is implemented. In Section 6.4, we outline how copies and versions of data objects are realized. Section 6.5 is devoted to the framework's basic and advanced concurrency control features. Section 6.6 describes how data objects are cached in our implementation. Section 6.7 provides details on transaction management and how data in Oracle databases is accessed. For performance reasons, our implementation is fully multi-threaded – the multi-threaded design is presented in Section 6.8. Section 6.9 describes our implementation of object and query routing. Section 6.10 discusses selected aspects of our implementation in more detail and, finally, Section 6.11 gives a summary of this chapter.

6.1 Overview

Our *Free Data Objects* framework has been implemented in the Java programming language, version 1.4, under the Linux operating system. The framework mainly consists of an implementation of an object manager component – at an architectural level, that component has already been briefly described in Section 5.1. The framework is accompanied by two command line tools that aid the development process (but their use is not mandatory for developers). The first tool generates a relational database schema from data object definitions. The second tool generates Java interfaces for type-safe access to data objects. At the time of writing, the source code of our FDO framework consists of about 24,000 lines of source code in about one hundred Java classes.

As has been shown in Figure 5-1, an object manager is placed in each process of the process topology of an enterprise application. Object managers of connected processes interact with each other and collaboratively fulfill data management functions for an enterprise application. Our object manager implementation is generic in the sense that it can be deployed in each process of a process topology. That means that the same implementation can be used in pure client processes as well as in processes with access to data stores or in processes that belong to intermediate tiers. Depending on its position in a process topology, an object manager will use all of its functionality or only a specific subset. The advantage of a generic implementation is that all object manager instances in a process topology are based on the same source code.

The framework has been named *Free* Data Objects mainly because we wanted to emphasize that instances of data objects (copies) are not restricted to one specific server process, as for example is the case with the distributed objects paradigm. Instead, copies can be freely propagated (copied by value) along C1 connectors in a process topology to any process that contains an object manager. As pointed out in Requirement R1 (see Subsection 4.4.1), this is essential for avoiding fine-grained communication and the resulting performance problems. Also, there is no need for application developers to use container classes for remote transfer and/or to duplicate server-side data object types (see Subsection 4.5.3). From a client/server database point of view, our implementation is based on the *object server* approach [DFMV90], i.e., (collections of) individual data objects are copied between clients and servers. Unlike *page servers*, which transmit entire disk/memory pages, the object server approach allows each object manager and each data store in the process topology to choose its own internal representation for caching/storing data objects. Particularly in heterogeneous topologies, this is advantageous.

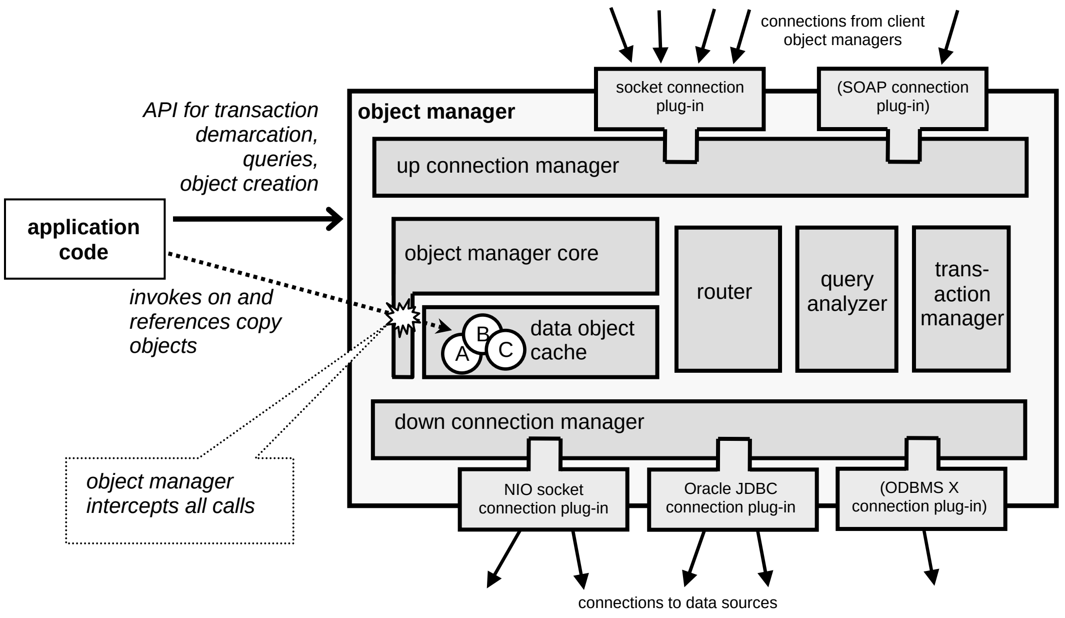

connections from client object managers
socket connection plug-in
(SOAP connection plug-in)
API for transaction demarcation, queries, object creation
object manager
up connection manager
application code
object manager core
router
query analyzer
trans-action manager
invokes on and references copy objects
B
A
C
data object cache
down connection manager
object manager intercepts all calls
NIO socket connection plug-in
Oracle JDBC connection plug-in
(ODBMS X connection plug-in)
connections to data sources

Figure 6-1. The internal architecture of an object manager component.

Figure 6-1 depicts the internal design of an object manager component. Application code that runs in the same process as the object manager may invoke functions on the object manager (e.g., to perform queries or to set transaction boundaries). Internally, an object manager maintains a *cache*, which contains copies of data objects. The copies are exposed to the application code, which may keep references to them and may invoke methods on them (typically *getXXX* and *setXXX* methods to access attributes). All invocations are transparently intercepted by the object manager, which ensures that each transaction “sees” an appropriate version of the accessed copy instance (see isolation described in Subsection 5.7.2) and that a copy is fetched from an underlying data source when a cache-miss occurs (see Section 5.6). The *router* component is responsible for routing items through the process topology (see Section 5.8). The *query analyzer* component, which is used by the router component, calculates which domains (see Subsection 5.2.2) are addressed by queries. The *transaction manager* component keeps track of all running transactions and the copies accessed by them. An object manager communicates with object managers of direct clients via the *up connection manager* component. With underlying data sources (object managers and/or transactional data stores), an object manager communicates via the *down connection manager* component. Into both connection manager components, protocol adapter implementations for different remote communication protocols can be plugged. Currently, two plug-ins for inter object manager communication through sockets and one plug-in for access to Oracle databases through JDBC are implemented. The plug-in mechanism makes it easy to add support for other protocols, e.g., the simple object access protocol (SOAP), protocols for accessing object databases (both shown in brackets in Figure 6-1), or CORBA.

Listing 6-1 depicts the signatures of the most important methods in the API that an object manager offers to application code (exceptions are not included).

```
public class ObjectManager {
   public ObjectManager( String name, String cfgString ) …
   public void release() …
   public void beginTx() …
   public Object suspendTx()…
   public void resumeTx( Object transactionHandle ) …
   public void rollbackTx()…
   public void commitTx()…
   public DataObject createNewDataObject( Class appInterface ) …
   public List query( String queryString ) …
}
```

Listing 6-1. The Java API an object manager exposes to application code.

To create an object manager component in a process, application code has to call the object manager’s constructor. A unique name for the new object manager and a string containing its configuration have to be passed as parameters. In the constructor, the object manager initializes itself, starts several background threads, and then is ready to serve the local application code as well as client processes. To shut down a running object manager, the *release* method has to be called. Application code can access data objects only within transactions, which are started with the *beginTx* method and terminated with the *rollbackTx* and *commitTx* methods. *beginTx* implicitly associates the calling thread with a newly created transaction (see introduction to Section 5.7). Internally, a *java.lang.ThreadLocal* object is used for that association. The methods *suspendTx* and *resumeTx* allow application code to temporarily disassociate and then re-associate threads with transactions, respectively. Within a transaction, application code can create new data objects by invoking the *createNewDataObject* method – the type of the new data object has to be passed as a parameter. Also, application code can execute queries, which return lists of references to data objects, via the *query* method.

The API of an object manager is similar to APIs typically offered by object databases and object/relational mapping tools. Listing 6-2 demonstrates how application code may use the API. We can see that access to transactional data objects is provided in a comfortable, transparent way – persistence, data distribution, remote communication, and other infrastructure details are transparently handled by the object manager.

```
String cfg = ...;                          // read cfg from file
ObjectManager localOM =                    // create object manager
   new ObjectManager( ”om17”, cfg );       //    with name “om17”
localOM.beginTx();                         // start new transaction
List orderList = localOM.query(            // execute query
   “SELECT FROM Order WHERE” +             //
   “reg=10 AND customer IS NOT NULL” );    //
Order order = (Order) orderList.get( 0 );  // get first result data object
Customer customer = order.getCustomer();   // navigational access
if( customer.getName().equals(”Meyer”) ) { // read a primitive value
   order.setDiscount( new Float( 0.06 ) ); // write a primitive value
}                                          //
localOM.commitTx();                        // commit transaction
// here we could perform some more txns
localOM.release();                         // shut down object manager
```

Listing 6-2. Example application code that demonstrates how the object manager API can be used.

The query language is similar to the query languages SQL [EN00] and OQL [CB00] but provides only a small subset of their features (in order to keep our proof-of-concept implementation relatively simple). Only sets of data objects can be selected. Projection, sub queries, and aggregation are not supported. A query string has the following form

```
SELECT FROM <type> [WHERE <condition>] [ORDER BY <attribute name> [ASC|DESC] ]
```

type must be the name of a data object type. *condition* is a Boolean expression that can be evaluated on the attribute values of a data object of the given *type*. Brackets, AND, OR, NOT, NULL, IS NOT NULL, compare operators, constant values, and attribute names may be used in the Boolean expression. Compare operators (>, >=, <, <=, <>, =) require an attribute name on the left side and a constant value on the right side (or vice versa). Attribute names may only be used in conjunction with compare operators. The order-by clause is optional and, if used, defines that the list of result data objects is to be sorted according to a given attribute. In addition to the queries described above, the FDO framework provides limited support for equi-joins[1].

6.2 Configuration

The FPT architecture states that the process topology and the data distribution scheme of an enterprise application are defined through configuration (see Section 5.4). In this section, we describe how configuration is realized in our FDO framework.

For simplicity, we will assume that (a) each process in a process topology is started manually[2] by an administrator and (b) that the configuration data for an object manager is stored in a local text file[3]. We recommend the following straightforward approach to start-up when using our framework:

1. The process is started. The object manager's unique name and the name of the configuration file are supplied as command line parameters to the process.
2. The application (start-up) code of the process reads the contents of the configuration file into a string.
3. The application code instantiates a local object manager as shown in Listing 6-2. The unique name and the configuration string are passed as parameters to the constructor.

An example configuration file is shown in Listing 6-3. A complete grammar of object manager configuration files can be found in Appendix A. In our implementation, an object manager parses its configuration data with code produced by the ANTLR parser generator [PQ95].

[1] For an equi-join query, two query strings and the name of a join attribute have to be supplied. Both query strings have to address the same domain. The result consists of two lists of references to data objects. The join is not a distributed join – only those pairs of objects are selected where both objects are stored in the same data store (pairs may come from different data stores). However, it would be possible to add a distributed join mechanism to the implementation.

[2] Alternatively, a system management tool or scripts could be used to automate the task of starting and shutting down processes on different machines in a network.

[3] In principle, the configuration can be stored in any place. The application code in each process is responsible for obtaining the configuration data and passing it as a string to the local object manager.

```
01: # Configuration file for object manager P4
02: OBJECT MANAGER P4 {
03:   ID = 6361;
04:
05:   DATA OBJECT Employee {
06:      TYPECODE = 200;
07:      empno:int; name:string; dep:Department;
08:      boss:Employee; retired:bool;
09:   }
10:   DATA OBJECT Department {
11:      TYPECODE = 201;
12:      name:string; address:string;
13:   }
14:
15:   INTERFACE FOR Employee:    empapp.dataobjects.Employee;
16:   INTERFACE FOR Department:  empapp.dataobjects.Department;
17:
18:   DOMAIN dom1 {
19:      CODE = 1001;
20:      DEF = SELECT FROM Employee WHERE retired=FALSE;
21:      RITREE = I(ds1, ds2);
22:   }
23:   DOMAIN dom2 {
24:      CODE = 1002;
25:      DEF = SELECT FROM Employee WHERE retired=TRUE;
26:      RITREE = ds2;
27:   }
28:   DOMAIN dom3 {
29:      CODE = 1003;
30:      DEF = SELECT FROM Department;
31:      RITREE = R(ds1, ds2);
32:   }
33:
34:   EXPORTS = (dom1, ds1) (dom1, ds2) (dom3, ds2);
35:
36:   SERVER CONNECTION {
37:      SERVERADDRESS = JDBCAddress
38:         "jdbc:oracle:thin@//foo.mi.fu-berlin.de:1521/MYDB"
39:         "clemens" "ah7tqk1y" "ds1";
40:      IMPORTS = (dom1, ds1);
41:      ADAPTER = OracleJDBCConnection;
42:   }
43:   SERVER CONNECTION {
44:      SERVERADDRESS = JDBCAddress
45:         "jdbc:oracle:thin@//bar.mi.fu-berlin.de:1523/XDB"
46:         "fdouser" "hjsb3epf" "ds2";
47:      IMPORTS = (dom1, ds2) (dom3, ds2);
48:      ADAPTER = OracleJDBCConnection;
49:   }
50:   CLIENT LISTENER {
51:      LOCALADDRESS = SocketAddress "snake.mi.fu-berlin.de" "12320";
52:      ADAPTER = SocketConnection;
53:   }
54: }
```

Listing 6-3. Example configuration of an object manager.

The example configuration file in Listing 6-3 is suitable, e.g., for the process *P4* of the process topology shown in Figure 5-6. The file covers all details outlined in Section 5.4: a subset of the enterprise application's process topology, a subset of the data distribution scheme, and a subset of the import/export scheme. Additionally, the configuration file contains definitions of data object types and interfaces. In the following, we explain the contents of the file in more detail:

- Each object manager is described in a separate *OBJECT MANAGER* section. An object manager reads only the section that matches its unique name. This allows multiple object managers to be configured within a single file. The example file contains only one *OBJECT MANAGER* section. That section configures an object manager named *P4* (lines 2 to 54),

- Lines 5 to 13 contain definitions of data object types. Two data object types are defined in the file: *Employee* and *Department*. Each type is described in a separate *DATA OBJECT* section, is assigned a unique type code, and has an arbitrary number of attributes. An attribute can be either a *bool*, *short*, *int*, *long*, *float*, *double*, *string*, or a reference attribute.
- In lines 15 and 16, data object types are associated with Java interfaces (which must be present in the local classpath). A Java interface for a data object type must extend the generic *DataObject* interface and should define a pair of get and a set methods for each attribute, for instance, *String getAddress()* and *void setAddress(String address)* for the address attribute of a data object type *Department*. Interfaces allow application code to access data objects in a type-safe manner. Associating data object types with interfaces is optional. However, a data object of a type that has no associated interface can only be accessed via the *DataObject* interface, which provides generic get and set methods, such as Object *getAttributeValue(int attributeIndex)* and *void setAttributeValue(int attributeIndex, Object value)*. More details on interfaces are given in Section 6.4.
- Lines 18 to 32 define a data distribution scheme as described in Section 5.2. Three domains – *dom1*, *dom2*, and *dom3* – are defined, each in a separate *DOMAIN* section. Each domain is defined through a query (or a sequence of coma-separated queries). For each domain, an RI-tree is defined (as text, in prefix notation), which describes how data objects of that domain are distributed among data stores. The RI-trees defined in lines 21, 26, and 31 correspond to the three RI-trees shown in Figure 5-6. An example of a more complex RI-tree was given in Figure 5-3 – in a configuration file, that tree would be textually represented as *I(R(ds2, ds3, I(ds5, ds6)), R(ds8, ds9))*.
- Line 34 defines the *exports* of process *P4*, i.e., which data objects may be imported and accessed by client processes of *P4* (see Section 5.3).
- The part of the process topology that is visible to *P4* is configured in lines 36 to 53. Each C1 connector from *P4* to a data source is defined in a separate *SERVER CONNECTION* section. Since *P4* has direct access to two data stores, its configuration contains two *SERVER CONNECTION* sections (lines 36 to 49); each specifies a protocol adapter plug-in to be used (JDBC access to Oracle), a remote host, a username, a password, and a data store name. In addition, each *SERVER CONNECTION* section defines what data is imported from the respective data source (see Section 5.3).

  C1 connectors from client object managers to *P4* are configured via *CLIENT LISTENER* sections. The *CLIENT LISTENER* section in lines 50 to 53 defines that *P4* accepts clients requests sent via plain sockets on port 12320. In contrast to a *SERVER CONNECTION* section, a *CLIENT LISTENER* section does not configure a single, concrete C1 connector but represents an arbitrary number of C1 connectors. Clients remain anonymous to *P4* until they actually request *P4*'s services at runtime.

In Listing 6-3, the configuration file contains only definitions for a single object manager. With such a scheme, each process in the process topology of an enterprise application is configured via a separate, locally available file. However, in some cases, it is desirable to have a single configuration file that contains configuration data for multiple object managers. For example, during development, a single configuration file can significantly simplify configuration in a single-machine or local area network environment. Or, when configuration data for a group of processes is created and managed in a central place, it is convenient to (manually or automatically) distribute the same single configuration file to all machines that run those processes.

The FDO framework supports multiple *OBJECT MANAGER* sections within a single configuration file. An object manager reads only the section that matches its unique name. Note that different object managers in a process topology often use similar definitions – especially definitions of data objects, interfaces, and domains. These common definitions can be placed in an *OBJECT MANAGER ** section. This reduces redundancy when multiple object managers are configured in a single configuration file. Listing 6-4 shows a configuration file that merges definitions for the object managers of two processes: *P2* and *P4* (both shown in Figure 5-6). Process *P2* is client of *P4*. Common definitions for both object managers are factored out into the *OBJECT MANAGER ** section. Also, we can see that *P2* defines a server connection to *P4* (lines 64 to 68) that matches *P4*'s client listener. *P2*'s configuration does not define any *EXPORTS* clause or *CLIENT LISTENER* sections because *P2* does not accept client requests from other processes.

```
01: # Configuration file for object managers P2 and P4
02: OBJECT MANAGER * {
...  (contains lines 5 to 32 from Listing 6-3)
31: }
32:
33: OBJECT MANAGER P4 {
34:   ID = 6361;
35:   EXPORTS = (dom1, ds1) (dom1, ds2) (dom3, ds2);
36:
37:   SERVER CONNECTION {
38:      SERVERADDRESS = JDBCAddress
39:         “jdbc:oracle:thin@//foo.mi.fu-berlin.de:1521/MYDB”
40:         “clemens” “ah7tqk1y” “ds1”;
41:      IMPORTS = (dom1, ds1);
42:      ADAPTER = OracleJDBCConnection;
43:   }
44:   SERVER CONNECTION {
45:      SERVERADDRESS = JDBCAddress
46:         “jdbc:oracle:thin@//bar.mi.fu-berlin.de:1523/XDB”
47:         “fdouser” “hjsb3epf” “ds2”;
48:      IMPORTS = (dom1, ds2) (dom3, ds2);
49:      ADAPTER = OracleJDBCConnection;
50:   }
51:   CLIENT LISTENER {
52:      LOCALADDRESS = SocketAddress “snake.mi.fu-berlin.de” “12320”;
53:      ADAPTER = NIOSocketConnection;
54:   }
55: }
56:
57: OBJECT MANAGER P2 {
58:   ID = 9371;
59:   SERVER CONNECTION {  # connects to P3 (not configured in this file)
60:      SERVERADDRESS = SocketAddress “worm.mi.fu-berlin.de” “12320”;
61:      IMPORTS = (dom1, ds1) (dom3, ds1);
62:      ADAPTER = SocketConnection;
63:   }
64:   SERVER CONNECTION {  # connects to P4
65:      SERVERADDRESS = SocketAddress “snake.mi.fu-berlin.de” “12320”;
66:      IMPORTS = (dom1, ds1) (dom1, ds2) (dom3, ds2);
67:      ADAPTER = NIOSocketConnection;
68:   }
69: }
```

Listing 6-4. Example file that configures multiple object managers. Common definitions are placed in the *OBJECT MANAGER ** section.

A configuration file contains all configuration data for one or more object managers. However, in some situations, administrators might not want to statically define all configuration data as part of configuration files. Instead, it might be more convenient to let a start-up script determine selected values (like IP addresses, port numbers, or object manager names) at runtime. With our implementation, this can be achieved by using *variables* in configuration files. For example, an administrator can change line 52 of the configuration file shown in Listing 6-4 to

```
LOCALADDRESS = SocketAddress ”$(MYHOSTNAME)” ”$(MYPORT)”;
```

and pass the values for *MYHOSTNAME* and *MYPORT* as command line parameters (using Java's *–D* option) when starting the process of the corresponding object manager. Especially when processes in a process topology are replicated, variables are convenient because the same configuration file can be used on different machines.

Configuration, as described in this section, plays a key role in the FDO framework. Through configuration, a software architect can construct arbitrary custom process topologies. Each object manager of a process topology is configured either by a separate configuration file or by a section in a common configuration file. This way, processes can be easily "plugged" together by configuring appropriate C1 connectors, i.e., matching pairs of *SERVER CONNECTION* and *CLIENT LISTENER* sections. Also, since the topology is not explicitly defined ("hard-coded") in the application code of an enterprise application, a software architect can easily adapt an existing custom process topology to changing requirements.

6.3 Communication between Object Managers

Only object managers connected via a C1 connector are allowed to directly communicate with each other. In the previous section, we learned how such communication links are configured. This section describes the internal implementation of inter-process communication.

As already mentioned in Section 6.1 and shown in Figure 6-1, an object manager uses its internal *up/down connection manager* components to communicate with directly connected client object managers and data sources. Into both connection manager components, different protocol adapters for remote communication can be plugged. Any combination of plug-ins is permitted – this allows an object manager to use different communication protocols for different remote object managers in parallel. A precondition for communication between a client object manager and a server object manager is that both are equipped with the same protocol adapter plug-in. Internally, the protocol adapter plug-ins are used for sending and receiving asynchronous point-to-point messages over the network. In principle, any underlying protocol (connection-based or connectionless) can be used to implement protocol adapter plug-ins. In the FDO framework, we have implemented two protocol adapter plug-ins for inter object manager communication:

- The *SocketConnection* plug-in is a simple plug-in that sends and receives serialized Java objects over plain sockets. For each connection to a client or server, an object manager maintains an open TCP connection and two dedicated threads (one for reading incoming data, the other for sending data).
- In addition, we implemented an optimized plug-in named *NIOSocketConnection*, which uses a more efficient wire format than Java object serialization and uses more scalable, low-level socket communication. The client side implementation is similar to that of the *SocketConnection* plug-in, except for the optimized wire format. The server side implementation is based on the new *nio* package introduced with Sun's JDK version 1.4. Under load, the implementation scales to many more concurrent clients than the *SocketConnection* plug-in because it uses a constant number of threads for IO (one thread per port) instead of two threads per client.

Within an object manager, the representation of a message is well-defined (by our framework). But how messages are represented on the wire (for instance, as XML formatted text, serialized Java objects, or CORBA/IIOP messages) is left to protocol plug-ins. Plug-ins are responsible for communication with remote sites and for translating between the object manager's internal message representation and the wire representation (marshalling and unmarshalling). The FDO framework defines twelve different message types – six pairs of request and reply messages. Communication always follows a strict client/server request-response style: Request messages are sent from client object managers to server object managers. For each request message received from a client, a server object manager sends back a corresponding reply message. Servers never proactively send messages to clients. As outlined in Subsection 5.9.2, this approach reduces complexity. The different message types used in our FDO framework are described in Table 6-1. Internally, each message type corresponds to a (serializable) Java class in our object manager implementation.

request message type	reply message type
A **ConnectRequestMessage** is sent from a client (e.g., on client start-up) to a server to initiate communication.	A **ConnectReplyMessage** informs the recipient whether its previous connection attempt has been accepted or rejected.
With a **QueryRequestMessage**, a client requests a server to return a set of copies of data objects. The client specifies the set either by including a SELECT query (see Section 6.1) or a reference to a particular data object to fetch.	As a reply to a query request, a server sends back a **QueryReplyMessage**. The message either indicates an error or contains a query result, i.e., a set of copies of data objects.
Messages of type **PushDownRequestMessage** are sent in the push-down phase of an FPT transaction (see Subsection 5.7.4). They are used for object routing (see Subsection 5.8.2), i.e., to propagate optimistic locks and private versions of committing FPT transactions "down" the process topology to transactional data stores.	A **PushDownReplyMessage** informs the recipient whether the routing it requested with a PushDownRequestMessage was successfully performed or led to an error. In the former case, the reply message includes information about where items have been routed.
PrepareRequestMessages are sent in the commit phase of FPT transactions (see Subsection 5.7.4) to access and then prepare transactional data stores that have received items in the push-down phase.	A **PrepareReplyMessage** informs the recipient about prepare votes of transactional data stores (o.k. or abort).
CommitRequestMessages correspond to the second phase of the two-phase commit (2PC) protocol. They are sent in the commit phase of FPT transactions to commit transactional data stores that have received items in the push-down phase and have been prepared with PrepareRequestMessages.	A **CommitReplyMessage** is sent back to an object manager to indicate a successful commit or an error.
A **RollbackRequestMessage** is similar to a CommitRequestMessage – however, it does not commit a transaction but is used to roll back a prepared transactional data store.	A **RollbackReplyMessage** is sent back to an object manager to indicate a successful rollback or an error.

Table 6-1. Pairs of message types defined by the FDO framework. Request messages are sent from client object managers to server object managers; reply messages vice versa.

Figure 6-2 shows an example message flow produced by an FPT transaction. The enterprise application's process topology and data distribution scheme are shown in the upper part of the figure. Each process *Pi* (*i*=1..5) of the topology contains a corresponding object manager *Oi*. Messages sent between object managers are shown in the UML collaboration diagram in the bottom part of the figure.

The example FPT transaction is initiated by application code in process *P1* that contains object manager *O1*. In its main phase, the transaction first queries a set of *Customer* data objects. As we can see from the left RI-tree, *Customer* data objects are ad-hoc distributed among data stores ds_2 and ds_3. Therefore, the set query has to be routed from *O1* to object managers *O4* and *O5*, which have access to ds_2 and ds_3, respectively. The query routing corresponds to messages 1, 1.1, and 1.2 in the diagram. After the query result has been delivered to *O1*, the FPT transaction (a) places an optimistic lock on the first result object and ignores other result objects, (b) creates a new *Order* data object, and (c) assigns values to its attributes. None of the three actions trigger any messages. Then application code in *P1* requests the local object manager *O1* to commit the FPT transaction. The transaction enters the push-down phase where the private version of the new *Order* data object is routed to data stores ds_1 and ds_2 and the optimistic lock on the *Customer* data object is routed to data store ds_2 (its home data store). The push-down phase corresponds to messages 2.1, 2.2, and 2.2.1 in the collaboration diagram. While message 2.1 carries one item (a private version), messages 2.2 and 2.2.1 both carry two items (a private version and a lock to validate). Finally, the FPT transaction makes a transition to the commit phase where a 2PC protocol is executed (with *O1* as coordinator). Prepare messages (3.1, 3.2, and 3.2.1) and commit messages (4.1, 4.2, and 4.2.1) are sent and follow the paths taken by the preceding push-down messages.

If the home data store of the locked *Customer* data object was ds_3 instead of ds_2, then a slightly different message flow would be the result. As before, the lock would be routed in message 2.2 together with the private version of the new *Order* data object. Then object manager *O3* would route the two items with two separate messages – one to *O4* and one to *O5*. In the collaboration diagram, we had to add three messages sent from *O3* to *O5* – 2.2.2: *PushDown*, 3.2.2: *Prepare*, and 4.2.2: *Commit*.

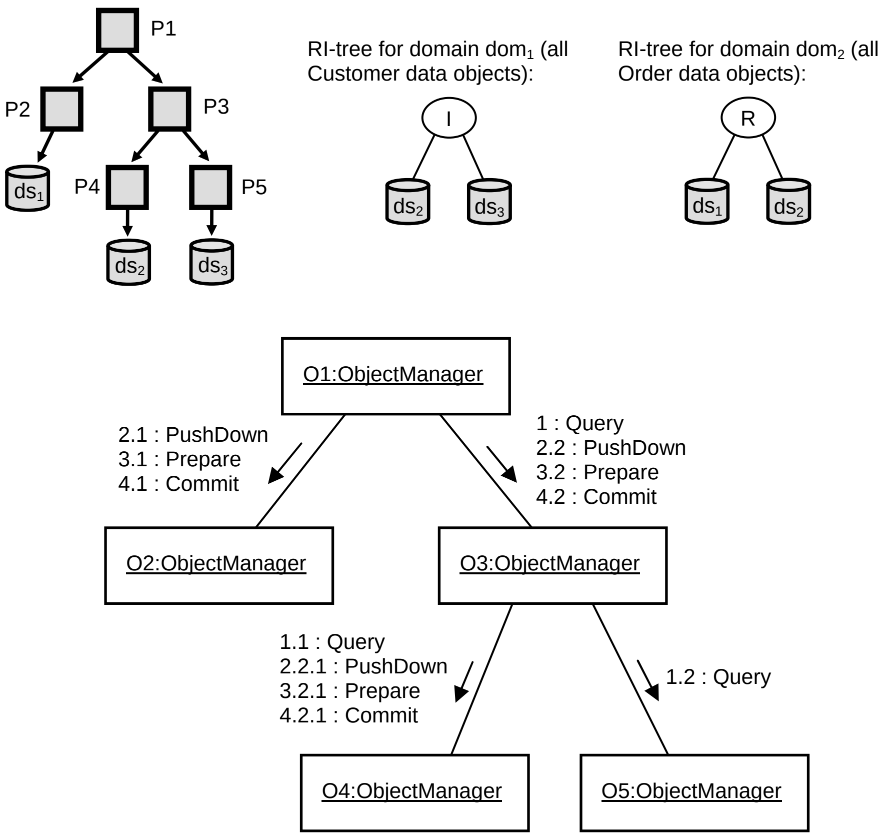


Figure 6-2. Collaboration diagram depicting messages of an FPT transaction that is initiated in *P1* (which contains object manager *O1*), queries *Customer* data objects, inserts a new *Order* data object, and finally commits.

Note that, for simplicity, the collaboration diagram uses the standard notation for sequential, synchronous messages. Technically, each message shown in the diagram maps to a pair of messages in the FDO framework (a request message from a client to a server and a reply message in the opposite direction, see Table 6-1). Also, the diagram shows all messages as a sequential flow. To reduce response time, our FDO framework sends multiple request messages in parallel whenever possible. More specifically, all messages with numbers that differ only in their last component (e.g., messages 2.1 and 2.2 or messages 1.1 and 1.2) are sent in parallel by an object manager. Internally, all messages are sent asynchronously – more details on tasks and multi-threading can be found in Section 6.8.

In this section, we outlined communication between object managers only. Communication between object managers and transactional data stores is covered in Section 6.7.

6.4 Copies and Versions

In Section 5.6 and Subsection 5.7.2, we described the general role of copies and versions in the FPT architecture – each object manager may cache data objects by keeping *copy* instances in its cache. To guarantee isolation, each copy instance may in turn contain multiple *versions*. In this section, we describe how copies and versions are implemented, how they can be accessed in a type-safe way, and how the association to their home data stores is maintained.

6.4.1 Type-Safe Access through Interfaces

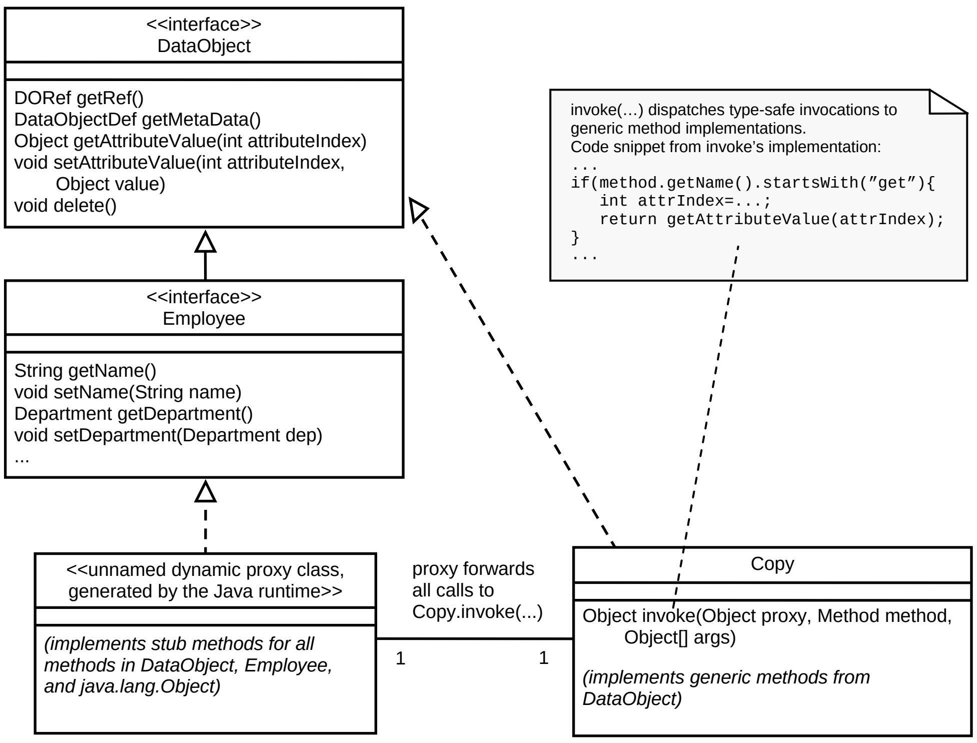
As described in Section 6.2, object managers can be configured to use given interfaces for data objects (e.g., lines 15 and 16 of Listing 6-3). With such specialized interfaces, application code always has a type-safe view on data objects and their attributes (see requirement R2 in Subsection 4.4.2). From a framework developer's perspective, it is always a crucial issue how user-defined interfaces/classes are linked to framework code in a transparent manner. In principle, this question has to be addressed in all frameworks that realize type-safe, transparent access to remote objects (e.g., CORBA or RMI) or persistent objects (e.g., object/relational mapping frameworks). In all such frameworks, application-specific code has to explicitly call framework code at some point. For instance, when application code resolves an object reference by calling the *getDepartment* method on an employee object, this may trigger a pointer swizzling mechanism [Moss92] in an object/relational mapping framework. Or, such a call may lead to network communication when the employee object is a remote CORBA object. On the one hand, framework code has to be invoked, but on the other hand, explicit calls potentially limit transparency. To preserve transparency, it is common practice to let a tool generate code that links application code to framework code. There are two typical approaches:

(1) Tools generate source code or byte code for complete classes that have to be used by application code – examples are stub and skeleton classes generated by CORBA IDL compilers or Java RMI compilers.

(2) Tools are employed that pre/post-process existing, user-defined code (source code or byte code) to insert ("inject") framework code. Examples are Java Data Objects (JDO) [JR03] and PJama [AJ00].

In our implementation, we provide type-safe, transparent access to data objects but do not require generation of framework code at compile time. We still need code that links application code to framework code, but that code is transparently generated at *runtime*. This is achieved through Java's *dynamic proxies*, which are generated on demand by the Java runtime[1]. The UML class diagram depicted in Figure 6-3 shows the relationships between interfaces, dynamic proxies, and framework code in our FDO framework.

[1] With prior versions of Sun's JDK, dynamic proxies – and reflection in general – introduced a significant overhead. Since version 1.4, these problems have been resolved.


<<interface>>
DataObject
DORef getRef()
DataObjectDef getMetaData()
Object getAttributeValue(int attributeIndex)
void setAttributeValue(int attributeIndex, Object value)
void delete()
<<interface>>
Employee
String getName()
void setName(String name)
Department getDepartment()
void setDepartment(Department dep)
...
<<unnamed dynamic proxy class, generated by the Java runtime>>
(implements stub methods for all methods in DataObject, Employee, and java.lang.Object)
proxy forwards all calls to Copy.invoke(...)
1
1
Copy
Object invoke(Object proxy, Method method, Object[] args)
(implements generic methods from DataObject)
invoke(...) dispatches type-safe invocations to generic method implementations.
Code snippet from invoke's implementation:

```
...
if(method.getName().startsWith("get"){
   int attrIndex=...;
   return getAttributeValue(attrIndex);
}
...
```

Figure 6-3. UML class diagram describing the use of dynamic proxy classes for realizing type-safe, transparent access to data objects (Java style signatures are used).

The *DataObject* interface, which is provided by the FDO framework, declares methods for generic (i.e., not type-safe) access to data objects. Attributes are addressed by their index number; attribute values are treated as *Object*s. Each copy of a data object is represented by an instance of the *Copy* class in an object manager. *Copy* is part of the FDO framework and implements the generic methods defined in the *DataObject* interface. For type-safe access, application developers may define specialized interfaces – one for each type of data object. In the class diagram, a specialized interface for data objects of type *Employee* is shown as an example. Application developers register specialized interfaces with an object manager by adding *INTERFACE FOR* statements to the object manager's configuration (see Listing 6-3, lines 15 and 16).

At compile time, there is no class that implements specialized data object interfaces. *Copy* contains the functionality that is needed but, technically, cannot directly implement the methods defined in the specialized interface. For linking specialized interfaces to the generic implementation in the *Copy* class, an object manager uses dynamic proxies. For each instance of *Copy*, an object manager requests the Java runtime to create a proxy instance via *java.lang.reflect.Proxy.newProxyInstance(...).* Between copy instances and proxy instances, there is a 1:1 relationship. A proxy instance belongs to a class that implements the given specialized data object interface – the *Employee* interface in our example. Proxy classes do not exist at compile time – instead, the Java runtime transparently generates appropriate classes that implement the specialized interfaces. Whenever an object manager passes a data object reference to application code (e.g., as part of a query result or as result of navigational access), the object manager does not pass a direct reference to the corresponding *Copy* instance but a reference to the instance's proxy. On the proxy instance, application code may invoke type-safe methods declared in a specialized interface because the proxy implements that interface. Proxies delegate all method invocations to their *Copy* instances; for each method invoked (e.g., *String getName()*), a proxy instance calls the *invoke* method on its corresponding *Copy* instance. The details of the invocation, i.e., the method and the parameters, are passed as parameters to the *invoke* method. As shown in the code snipped in the class diagram, the implementation of *Copy.invoke(...)* examines its parameters and calls the appropriate generic method, e.g., *Object getAttributeValue(int)* for method *String getName()*. Application code never directly references instances of *Copy* – an object manager makes sure that a proxy is always passed to application code instead. For data object types for which application developers do not provide specialized interfaces, the proxy directly implements the generic *DataObject* interface.

The approach described above is an elegant way of providing the interception functionality described in Section 6.1. The approach has the advantage that application code and framework code are strictly separated – unlike conventional approaches, which add generated framework code to the application code. Without a clear separation, changes to a framework might break existing, already deployed enterprise applications because their generated code parts remain unchanged. Also, our approach is simple and based on standard Java APIs; no modifications to the Java virtual machine and no low-level byte code modifications are necessary.

Application developers that use our FDO framework may manually define interfaces for type-safe access to data objects. In that case, no separate code generation step is necessary at compile time. Alternatively, developers can use our *InterfaceGenerator* tool that reads an object manager's configuration file and generates a Java interface (source code) for each data object type defined in the configuration. Using our tool introduces a compile time code generation step like many other frameworks do. However, the separation of application code and framework code is preserved because only interface definitions (which belong to the application code) and not a single line of framework code are generated.

6.4.2 Implementation of Copies and Versions

In the previous section, we outlined that, at runtime, each cached data object is represented by a pair of object instances in our implementation: a *Copy* instance and a dynamic proxy instance for forwarding type-safe invocations to the *Copy* instance. To guarantee isolation, each *Copy* instance may in turn contain multiple *versions* as described in Subsection 5.7.2. There are *public versions*, which reflect snapshots of data objects at different times (committed values only), and *private versions*, which store values not yet committed by a transaction. In the following, we explain how copies and versions are realized in our implementation.

Table 6-2 lists the most important instance variables (all declared as private) defined by the *Copy* class.

instance variable	description
`DataObjectDef md`	Contains metadata for the type of the data object represented by this copy instance, including the number of attributes, their names, and their types.
`Object myProxy`	Reference to the dynamic proxy instance that proxifies this copy.
`DORef myRef`	Data object reference of this (i.e., to this) copy. A *DORef* instance contains a unique object identifier (64 bit), a data object type code (16 bit), a domain code (16 bit), and a bit field that stores the data object's home data stores (32 bit).
`short lifeCycleState`	A copy is always in one of these four states: *LCS_CREATED*, *LCS_ALIVE*, *LCS_DELETED*, *LCS_CREATED_DELETED*
`PublicVersionSet publicVersions`	Stores all public versions contained in this copy. Each public version corresponds to an instance of class *PublicVersion*.
`HashMap txToAccessState`	Map that contains one entry for each running transaction that has accessed this copy. Each such transaction is mapped to an *AccessState* instance (which stores access information for the transaction such as lock status, public version seen, and possibly a private version).
`long lastSyncTime`	Absolute time (in ms) this copy has last been synchronized with its persistent state. Serves as a freshness indicator – see Section 6.6 for details.

Table 6-2. Private instance variables of the *Copy* class.

In this subsection, we describe the three variables *lifeCycleState*, *publicVersions*, and *txToAccessState* in more detail.

Variable lifeCycleState

Each *Copy* instance always is in one of four life cycle states – a corresponding state value is stored in the *lifeCycleState* instance variable. The UML statechart diagram shown in Figure 6-4 depicts life cycle states and state transitions.

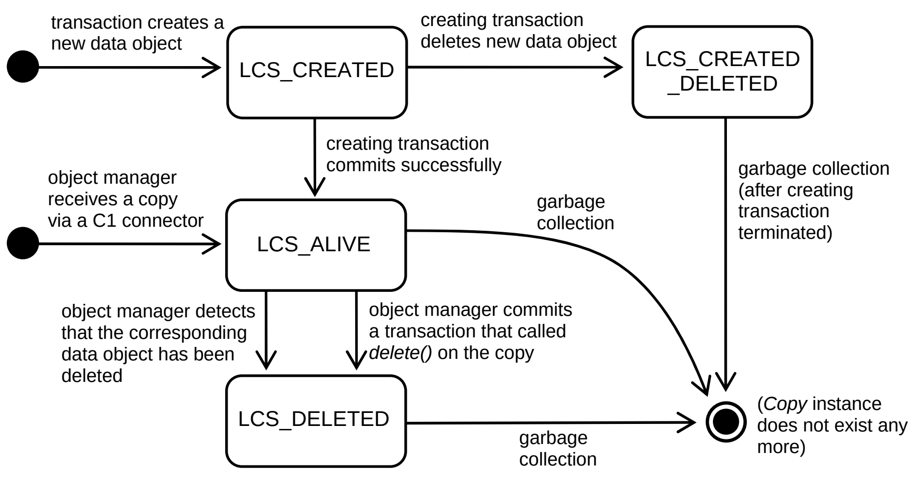

transaction creates a new data object
LCS_CREATED
creating transaction deletes new data object
LCS_CREATED _DELETED
creating transaction commits successfully
object manager receives a copy via a C1 connector
LCS_ALIVE
garbage collection
garbage collection (after creating transaction terminated)
object manager detects that the corresponding data object has been deleted
object manager commits a transaction that called *delete()* on the copy
LCS_DELETED
garbage collection
(*Copy* instance does not exist any more)

Figure 6-4. A statechart diagram describing life cycle states of a *Copy* instance and transitions between states.

When an FPT transaction creates a new data object, the *lifeCycleState* of the corresponding *Copy* instance is set to *LCS_CREATED*. In that state, the *Copy* instance is only visible to the FPT transaction that created it and does not contain any public versions yet. Only after a successful commit, the *Copy*'s *lifeCycleState* is set to *LCS_ALIVE* and a first public version is inserted into it. Also, when an object manager loads a copy from a server, that copy is set to *LCS_ALIVE*.

In state *LCS_ALIVE*, any number of concurrent FPT transactions can read and update the corresponding data object as described in Subsection 5.7.2. Also, the *delete()* operation can be called by an FPT transaction, which creates a new private version (with a delete marker) in that copy. On successful commit of an FPT transaction that deleted a data object, the data object's persistent state is removed from all data stores and *lifeCycleState* of the copy is set to *LCS_DELETED*. When an object manager sets a *Copy*'s *lifeCycleState* to *LCS_DELETED* (either because the object manager commits a delete operation or because it detects that an FPT transaction in another object manager deleted the corresponding data object), only running FPT transactions that already have accessed the *Copy* until then may continue to access the *Copy*. This ensures repeatable reads even when a data object has been deleted. It is also possible that a data object has been deleted but some object managers in the process topology still contain stale copies of that data object in their caches because they have not detected the delete yet. In that case, their (stale) *Copy* instances are still in state *LCS_ALIVE* and can be accessed by local FPT transactions. However, updates, deletes, and the validation of optimistic locks will fail during the commit phase of such FPT transactions (see Subsection 5.7.4). State *LCS_CREATED_DELETED* is only used when a data object is created and deleted within the same transaction (and thus never becomes visible to other transactions).

Consider that an object manager *OM* commits a transaction *T* that has created, updated, or deleted a data object *D*. There is a small chance that, between the execution of commit on underlying data stores and the arrival of the commit notification at *OM*, other transactions work on the newly committed persistent state of *D*. Thus, it is possible that *OM* detects the changes caused by *T* or by other transactions before it receives a *CommitReplyMessage* for *T*. To handle such cases, there are two additional state transitions which are not shown in Figure 6-4: one from *LCS_CREATED* to *LCS_ALIVE* and one from *LCS_CREATED* to *LCS_DELETED*. Those transitions can occur, e.g., when a *CommitReplyMessage* gets lost or delayed or when messages are not processed in the order they have been received. Intentionally, sending and processing messages in a specific order is neither required nor guaranteed with our implementation.

Variable publicVersions

The *publicVersions* instance variable (see Table 6-2) holds a reference to a set of (one or more) public versions of the data object. Each public version corresponds to an instance of class *PublicVersion*, which defines two instance variables: *int counter* (to distinguish different public versions; see Subsection 5.7.2) and *Object[] values* (which stores zero or more attribute values).

The set does not reference *PublicVersion* instances contained in it with standard (hard) Java references. Instead, Java's *weak reference* mechanism is used so that outdated public versions are removed when they are no longer needed. Once there is no hard reference to a given *PublicVersion* instance any more, the instance becomes eligible for garbage collection. To prevent the most recent public version contained in the set from being garbage collected, the set maintains an additional hard reference to the public version with the highest counter value. With weak references as described above, an object manager typically stores only one or two public versions per copy.

Variable txToAccessState

A *Copy* instance cached in an object manager can be accessed by multiple concurrent transactions, which may work on different public and private versions. To keep track of concurrent transactions, a *Copy* instance maintains the *txToAccessState* map, which stores an *AccessState* instance for each running transaction that has accessed the *Copy* instance. An *AccessState* instance stores access information for a transaction *T*. This includes

- the type of access, which is one of the following: (a) *T* has newly created the *Copy* instance, (b) read-only access, (c) update access, and (d) *T* has performed a *delete* operation on the instance.
- for access types (a) and (c), a private version, which stores all attribute values written by *T*.
- a (hard) reference to a public version. For access type (a), the reference is set to *null*. For (b), the reference points to the public version seen by *T*. For (c), the reference points to the public version on which the private version (see above) is based. For (d), the reference points to the public version seen by *T* before *T* performed the *delete* operation on the instance.

With the data structures and logic described in this subsection, a *Copy* instance keeps track of its state and guarantees isolation of concurrent transactions. Each operation performed by a transaction on a data object (e.g., *create*, *getXXX*, *setXXX*, *delete*) is handled by the corresponding *Copy* instance, which provides each transaction with an appropriate view on data. For application code, the management of *Copy* instances, their internal data structures, and isolation logic is transparent. It is even safe to pass a reference to a data object from one thread/transaction to another thread/transaction. Although there may be different versions for different transactions, a data object appears as a single Java object instance to all transactions/threads in a process, i.e., object identity is preserved (see requirement R2 in Subsection 4.4.2).

We included neither inheritance nor user-defined methods for data objects in our proof-of-concept implementation. Both features are convenient for application developers but are not essential from an architectural point of view. However, it is possible to extend our implementation to support inheritance and user-defined methods; no major redesign is required in that case.

6.4.3 Home Data Stores

In Section 5.2, we described the FPT architecture's approach to data distribution. Each data object is stored in one or more *home data stores*. A home data store of a data object stores a replica of the data object's persistent state. For a newly created data object, the middleware framework selects a set of home data stores according to the enterprise application's data distribution scheme. When the corresponding RI-tree for a data object contains I-nodes (ad hoc distribution), the middleware has multiple options for selecting home data stores.

Ad hoc distribution has several advantages (see Subsection 3.4.4). However, in a middleware implementation, care must be taken that data management remains efficient when a data distribution scheme permits multiple options. To ensure good performance, we made the following two important design decisions:

1. *The set of home data stores of a data object is stored as part of the object's persistent state and as part of each copy.*

   Consider, for instance, that the FDO framework has selected ds_1, ds_4, and ds_5 as home data stores for a data object *D* and thus *D*'s persistent state is replicated to all three data stores. In addition to *D*'s attribute values and its persistent counter value (see Subsection 5.7.2), each of the three data stores contains the information that *D*'s home data stores are ds_1, ds_4, and ds_5. Also, that information is stored within each copy of *D* cached within an object manager.

2. *The set of home data stores of a data object is encoded into each object reference to that data object.*

   There are two types of attribute values: primitive values and object references (to other data objects). Like all attribute values, object references are stored within public and private versions as well as within the persistent state(s) of a data object. A simple implementation would just use the unique object identifier (see Section 5.6) of a data object *D* to encode object references pointing to *D*. However, in the FDO framework, we also store the set of *D*'s home data stores as part of the object reference.

Both design decisions introduce a certain amount of redundancy and slightly increase the size of data objects in data stores as well as in main memory. At the same time, that redundancy can be used to achieve significantly better performance when updating and deleting data objects and for navigational access. We illustrate that with the help of two examples:

Example 1: Application code in a process *P* executes a set query for customer data objects from a domain for which the RI-tree given in Figure 5-3 has been defined. Various data stores are queried (see query routing in Section 5.8) and, for each result object, a copy is created in *P*'s object manager. One of the copies (for data object *C* with home data stores ds_2, ds_3, and ds_5) is loaded from data store ds_3. Then the application code updates one of *C*'s attribute values and commits. To execute the commit request, the middleware framework has to change *C*'s persistent states in ds_2, ds_3, and ds_5 (ROWA replication). Since *C*'s home data stores are known to *P*'s object manager (because of the first design decision), updates can be directly routed from *P* to *C*'s home data stores.

If home data stores were not stored as part of persistent states and copies, *P*'s object manager could possibly know that ds_3 is one of *C*'s home data stores (because the copy was loaded from ds_3) and, consequently, ds_2, too (by analyzing the RI-tree). However, the object manager could not know whether ds_5 or ds_6 is *C*'s third home data store. To comply with ROWA replication, updates to *C* either had to be send to *all* data stores that are potential home data stores of *C* (ds_2, ds_3, ds_5, and ds_6 in this case) or expensive probe queries had to be performed to identify all home data stores first.

Example 2: The object manager in a process *P2* caches a copy of an order data object *O*. *O* has an attribute named *customer* that contains an object reference to a customer data object *C* (which is the same data object as described in the first example). Application code in *P2* calls the *getCustomer()* method on *O*, i.e., performs a navigational access. Since *C*'s home data stores are encoded into the object reference (see second design decision), an instance query (see Section 5.8) can directly be routed from *P2* to any one of *C*'s home data stores. If object references consisted of object identifiers only, the instance query had to be routed to multiple data stores (which is less efficient) in order to find one data store that stores *C*.

In our implementation, a set of home data stores is stored as a bit field of fixed-length (32 bit)[1]. Each bit represents one data store in the process topology. The advantage of a bit field representation is that most set operations (e.g., insert, intersection, minus) can be performed extremely efficiently in terms of processor cycles and memory usage.

6.5 Basic and Advanced Concurrency Control

Section 5.7 described the FPT architecture's approach to transaction management and concurrency control. Our implementation realizes those concepts. In addition, the implementation includes several advanced concurrency control features that go beyond the (minimum) requirements of the FPT architecture. In this section, we outline how basic and advanced concurrency control features are implemented in the FDO framework.

[1] Another (even more space-efficient) way would be to assign a number to each insert option permitted by an RI-tree and store that number. Alternatively, variable-length bit fields could be employed instead of fixed-length bit fields.

6.5.1 Check Types

The FPT architecture relies on optimistic concurrency control – on each access to an existing data object (*update*, *delete*, call to a *getXXX* method), an optimistic lock is placed on that data object. During the commit phase of an FPT transaction, all optimistic locks are validated (using version counter values). When at least one optimistic lock cannot be evaluated, the FPT transaction is aborted. This approach is relatively simple and thus suitable for describing the architecture of a middleware framework. However, as already discussed in subsections 5.7.5 and 5.9.1, that approach also is relatively conservative because many locks are granted, which have to be validated and, in some cases, might produce unnecessary transaction aborts. To address that problem, our implementation uses three types of optimistic locks ("checks") instead of only one[1]:

- An *existence check* is the weakest check; it is successful as long as the corresponding data object has not been deleted. The following example explicitly adds an existence check to a given account data object:

  ```
  account.addExistenceCheck();
  ```

- A *predicate check*[2] evaluates a given Boolean expression on the current committed persistent state of a given data object. The expression is evaluated on a single data object only, i.e., cannot be applied to sets of data objects. A predicate check is only successful when the given expression is evaluated to *true*. A predicate check always includes an existence check and thus is more restrictive. The following example explicitly adds a predicate check to a given account data object:

  ```
  account.addPredicateCheck( ″balance>=0″ );
  ```

- A *version check* corresponds to the optimistic lock described in Section 5.7. A version check on a data object is only successful if the corresponding FPT transaction (still) works on a fresh version of the data object when the FPT transaction commits. A version check includes an existence check. However, a version check generally does not include a predicate check or vice versa. The following example explicitly adds a version check to a given account data object:

  ```
  account.addVersionCheck();
  ```

All checks are added while an FPT transaction is running, but checks are performed only during the commit phase of an FPT transaction. Internally, all checks are stored within *AccessState* instances (see Subsection 6.4.2).

For pessimistic locks, it is common to define a compatibility matrix. However, since checks never conflict with other checks, a compatibility matrix does not make sense here. Instead, Table 6-3 shows what type of check protects an FPT transaction against what phenomena (by aborting the FPT transaction in its commit phase when a phenomenon is detected).

[1] The term *lock* can be misleading because it suggests that a transaction is blocked when it accesses a data object that is already locked by a concurrent transaction. However, that is only the case with pessimistic locks; optimistic locks are validated on commit and do not block concurrent transactions. Although the term *optimistic lock* is frequently used in the literature, we decided to use the term *check* for more clarity when discussing FDO implementation details. We say that FPT transactions *add* checks to data objects.

[2] Not to be confused with *predicate locks*, which pessimistically lock sets of items in a database.

	T1 accesses a stale copy of D (e.g., because of caching)	D was updated by a concurrent transaction T2	D was deleted by a concurrent transaction T2
existence check	not detected	not detected	detected
predicate check	depends on predicate	depends on predicate	detected
version check	detected	detected	detected

Table 6-3. Transaction T1 accesses a data object D and adds a check to it. The table shows what phenomena are detected by what types of check.

6.5.2 Delta Writes

Until now, we have assumed that updates to a data object are always made via standard *setXXX* calls that overwrite an attribute value with a given, absolute value. In our implementation, we provide an additional update operation that *changes* an attribute value (relative to its previous value) instead of overwriting it. We call such updates *delta writes*[1]. In contrast to standard updates, which are general purpose, delta writes are a semantics-based approach, see Subsection 5.9.1.

For example, with standard update operations, a credit transfer could be implemented as shown below:

```
float bal1 = account1.getBalance().floatValue();
float bal2 = account2.getBalance().floatValue();
account1.setBalance( new Float( bal1 – 100.0 ) );
account2.setBalance( new Float( bal2 + 100.0 ) );
```

In contrast to that, an implementation that uses delta writes would update the account objects as follows:

```
account1.setAttributeValue( 3 /*balance*/, new DeltaWrite(”-100.0”) );
account2.setAttributeValue( 3 /*balance*/, new DeltaWrite(”+100.0”) );
```

When an FPT transaction updates an attribute value and then reads it again with a *getXXX* call, always the updated value is returned – for standard updates as well as for delta writes (repeatable reads are guaranteed). A delta write applied to a data object with a version check is equivalent to a standard update to a data object with a version check. However, for existence and predicate checks, the behavior is different. In the commit phase, a standard update writes an absolute value, whereas a delta write is recalculated and applied to the current persistent state found in the data store(s).

For instance, assume that two FPT transactions *T1* and *T2* concurrently deduct 100.0 from the balance attribute value of an account data object *D*. First, *T1* and *T2* query *D* and read its balance (500.0). Then both transactions add an existence check to *D* and update *D*'s balance value. Finally, both transactions re-read *D*'s balance again and commit. With standard updates, both transactions commit successfully (no conflict since no version check has been used), but *D*'s persistent state stores a final balance value of 400.0 instead of 300.0 (lost update). With standard updates and version checks instead of existence checks, only one of the transactions can commit successfully; the other is aborted. However, with delta writes and existence checks, both transactions commit successfully and the final balance value is 300.0, as desired. Note that, when the transactions re-read *D*'s balance, both read a value of 400.0 – for standard updates as well as for delta writes. Nevertheless, on commit, delta writes are recalculated based on the persistent state found in the data store.

[1] Currently, only addition and subtraction to/from numeric values are implemented. However, in principle, all commutative operations could be used.

6.5.3 Automatic and Manual Check Mode

The FDO framework's default mode is the *automatic check mode*. In this mode, application code does not explicitly add checks to data objects. Instead, the framework transparently adds checks to data objects as they are accessed. While the automatic check mode is convenient for application developers and provides a relatively high level of consistency, it provides only standard concurrency control features. To benefit from advanced concurrency control, application code may switch to the framework's *manual check mode*. In this mode, only a minimum level of consistency is guaranteed – for more consistency, application code has to explicitly add more restrictive checks. The check mode is set separately for each application thread. A thread can change the mode anytime, even while a transaction is running. For example, when *om* references the local object manager instance, an application thread can switch to manual check mode as follows:

```
om.getThreadContext().setAutoCheckMode( false );
```

Table 6-4 shows which checks are implicitly added by the FDO framework in automatic check mode and manual check mode. For example, when an FPT transaction deletes a data object while in automatic check mode, the framework implicitly adds a version check to the data object. In both check modes, application code is free to explicitly add more restrictive checks any time[1].

	read (getXXX)	**update (setXXX)**		**delete**	**insert**
		standard	**delta write**		
no check	A, M	▨	▨	M	A, M
existence check		M	A, M		▨
predicate check		*	*		▨
version check		A		A	▨

A: check implicitly added in automatic check mode
M: check implicitly added in manual check mode
* : combinations that correspond to *field calls*

Table 6-4. Combinations of access operations and checks. Combinations that are not permitted/possible are shown in gray.

In addition to the checks shown in the table, the FDO frameworks implicitly adds an existence check to the *referenced* data object when an attribute of type reference is written. This is done in automatic check mode only and prevents referencing a data object that is deleted by a concurrent transaction.

Note that, in either mode, the framework does not add any checks to data objects that have only been read by the FPT transaction. That behavior corresponds to a lower level of consistency than assumed by the FPT architecture. To guarantee the level of consistency assumed by the FPT architecture, application developers have to explicitly add version checks to such data objects. Our implementation does not add checks in that case because, typically, such checks are not required by an enterprise application. For example, consider an FPT transaction that queries a set of order data objects, reads their attribute values to display them to a user, updates one of the order objects (update of the *shippingMethod* attribute value),

[1] It is permitted to add multiple checks (even of different type) to the same data object. Internally, an existing check and a newly added check are immediately combined into a single check. For example, an existence check and a version check are combined into a version check. Or, a predicate check with Boolean expression *expr1* and a predicate check with Boolean expression *expr2* are combined into a predicate check with Boolean expression *expr1* AND *expr2*. A predicate check and a version check are combined

and finally commits. In this case, only a single check (for the updated order data object) is required – applying checks to all order data objects read is neither required nor efficient.

In addition to the checks added implicitly by the framework, Table 6-4 also shows two combinations that are of particular interest because they correspond to *field calls* [GR93]. Field calls are updates that are only applied if a given predicate is evaluated to *true*. Field calls have been implemented in IBM's Information Management System (IMS) Fast Path, where they correspond to *FLD/Verify* and *FLD/Change* calls [IBM03b]. Note that field calls as realized in IMS Fast Path and described in [GR93] are implemented at the database level, whereas our field calls are implemented at the framework level. However, the basic mechanism remains the same.

6.5.4 Advantages

Concurrency control, as implemented in our FDO framework, has several advantages:

- In most cases, application developers can rely on the automatic check mode, which is convenient because concurrency control is transparently managed by the framework.
- The framework gives application developers the flexibility to execute FPT transactions with a lower level of consistency (typically, to improve performance) or a higher level. This can be achieved through the framework's manual check mode and/or checks that are explicitly added. We expect that only selected parts of some enterprise applications require the framework's advanced concurrency control features. Application code can rely on standard concurrency control and, for selected operations, switch to manual check mode if necessary.
- There is a broad range of options when using advanced concurrency control features (see Table 6-4). If necessary, data access operations can be *fine-tuned on a per-object basis* to achieve a good trade-off between consistency and performance.
- The advanced concurrency control features include the functionality of field calls. Especially delta writes in combination with predicate checks (or existence checks) have two important advantages over standard updates: (1) They allow for more concurrency and thus higher transaction throughput. The risk of transaction aborts due to updates of concurrent transactions is greatly reduced. (2) Caching can be much more effective. In many typical cases, transactions can operate even on cached (and thus possibly stale) hotspot data objects. An example of field calls and a more detailed discussion can be found in Section 7.7.

6.6 Object Caching

For the FPT architecture, caching has been described in Section 5.6. In addition, several caching related details have been discussed in Section 5.7. In this section, we outline how caching is implemented in the FDO framework.

The framework performs caching completely transparent to application code. All copies of data object created within or loaded into an object manager are stored into the object manager's object cache. The cache stores and looks up copies of data objects by their unique object identifier, which is a Java *long* value. Internally, the cache is implemented as a hash table that hashes object identifiers to *Copy* instances[1]. All local transactions work on the same (shared) cache of an object manager. The cache uses a least-recently-used scheme for eviction.

into a version check; additionally, the predicate is immediately evaluated on the version currently seen by the FPT transaction and the transaction is set to *rollback-only* if the evaluation result is *false*.

[1] To keep the implementation simple, we implemented only basic object caching. The object cache speeds up navigational access but does not remember which copies belong to which query result. To improve the performance of queries, a query caching mechanism had to be added to the implementation (see Section 8.2).

In many object caching systems, a *pointer swizzling* mechanism [Moss92] is employed to convert virtual references to main memory references and vice versa. That is not the case in our implementation. Attribute values in copies that represent references to other data objects are always represented in the same way (as an instance of *DORef*), whether a copy of the referenced data object is present in the cache or not. A *DORef* instance does not contain a Java reference – instead, it contains the referenced object's unique identifier, which can be used to lookup the referenced data object. The advantage of that approach is that copies can be loaded into and evicted from the cache without updating references that point to them. On the other hand, each navigational access always requires a cache lookup with our approach. However, since cache lookups are based on hashing and thus are sufficiently fast, that is acceptable.

To prevent a cache from growing too large, a cache eviction mechanism is typically employed. In our implementation, we rely on Java's *soft references* and the Java virtual machine's garbage collector for eviction. The object cache itself does not maintain any hard Java references to *Copy* instances. Instead, Java *soft references* are used. Once there is no hard reference to a *Copy* instance any more, the instance becomes eligible for garbage collection. When the garbage collector claims the instance, it is removed from the cache. Note that there is an important difference between weak references (used for referencing public versions, see Subsection 6.4.2) and soft references. Objects that are reachable only via weak references are to be claimed rather sooner than later. In contrast to that, virtual machine implementations are supposed not to claim objects reachable only via soft references when they have been recently used [CLK99]. The existence of hard references to a cached *Copy* instance intentionally prevents it from being garbage collected. Hard references can either be held by application code (indirectly via a dynamic proxy instance, see 6.4.1) or be stored within a *Transaction* instance. A *Transaction* instance represents a running transaction and maintains hard references to all copies accessed by the transaction.

When copies of data objects are cached, it might be necessary to prevent the copies from becoming too stale (because working on stale data might prevent transactions from committing successfully). To give application developers some control over the freshness of cached data, we have implemented three different cache policies:

1. The *NOCACHING* policy prevents caching. Each access of an FPT transaction to a data object forces the local object manager to load/reload a fresh copy from one of the data stores (directly or indirectly via other object managers). However, after a copy of a data object has been accessed for the first time by an FPT transaction, further accesses of the same transaction always work on the local copy to guarantee repeatable reads. The *NOCACHING* policy should be used for maximum freshness but typically requires more remote interactions.
2. With the *FRESHNESS* policy, copies of data objects are cached but the object manager attempts to keep them relatively fresh (according to a *freshness* parameter, which defines a desired minimum freshness in milliseconds). For each *Copy* instance, the object manager remembers the time of the last synchronization with the data object's persistent state (see *lastSyncTime* in Table 6-2). Both read and write access count as synchronization. When an FPT transaction accesses a cached copy for the first time, the object manager forces a reload when *lastSyncTime+freshness* < *currentTime*. The *FRESHNESS* policy is the default cache policy; the *freshness* parameter defaults to 3,000 milliseconds.
3. The *UNLIMITED* policy does not trigger reloads and allows transactions to access arbitrarily stale data. The *UNLIMITED* policy should only be used if freshness is less important or the cached data is never or rarely changed.

Application code can change the cache policy and the *freshness* parameter any time, even while a transaction is running. For example, application code may use the *FRESHNESS* cache policy for most operations and temporarily switch to the *NOCACHING* policy for accessing selected crucial data objects. The cache policy is set per object manager and thread to avoid conflicts between concurrent transactions/threads.

6.7 Database Access and Transaction Management

In principle, any type of (XA compliant) transactional data store can be integrated into our FDO framework. As outlined in Section 6.1, a separate plug-in is necessary for each type of data store. For application code, the number and types of underlying data stores are transparent – database access is managed by the framework. As part of our proof-of-concept implementation, we provide a plug-in for the Oracle 9i relational database management system. In this section, we outline how data objects are mapped to relational data by the plug-in, give details on how relational data is accessed, and describe how the framework manages distributed transactions.

6.7.1 Object/Relational Mapping

Our *OracleJDBCConnection* plug-in implementation relies on Oracle's JDBC driver to access Oracle 9i databases. The plug-in implements a simple object/relational mapping as follows. Each data object type is mapped to a dedicated table. Each data object instance is mapped to a row in such a table. A table always has at least the following columns:

- The *OID* column (primary key) is of type *DECIMAL(20)* and stores a data object's unique object identifier[1] (see Section 5.6). By default, Oracle creates an index for this column.
- The *INFO* column is of type *DECIMAL(20)* and stores a data object's type code, domain code, and home data stores (see *DORef* in Table 6-2 and Subsection 6.4.3).
- The *VERSION* column is of type *INTEGER* and stores a data object's persistent counter (see Subsection 5.7.2), i.e., the number of the most recent committed version.

For each primitive attribute value of a data object, a column of an appropriate type is added to its table. For each object reference attribute, two columns are added; one stores the unique object identifier of the referenced data object (i.e., a foreign key) and the other stores the same data as the *INFO* field stored for the referenced data object (that data helps to locate the referenced object when it is stored in another data store, see Subsection 6.3.4). To simplify development, we have implemented a command line tool (*DBSchemaCreator*) that generates an initial relational database schema from the data objects defined in a given object manager configuration file.

Intentionally, we kept the object/relational mapping simple in our proof-of-concept implementation. As already pointed out in Subsection 6.4.2, inheritance is not supported. However, it is straightforward to realize additional plug-ins that support more sophisticated and also custom mappings (e.g., by using advanced object/relational mapping frameworks for that particular task). Furthermore, our object/relational mapping component makes use of relational features only. For datastores with object support (object or object-relational DBMS), additional optimizations are possible.

6.7.2 Access to Relational Data

Relational data in an Oracle database is accessed (a) when an FPT transaction performs a query that cannot be answered by a cache and (b) during the commit phase of an FPT transaction (see Subsection 5.7.4). The handling of queries is straightforward; framework queries are directly translated into SQL SELECT queries that return all result data objects (one per row). The actions executed during the commit phase of an FPT transaction depend on the data objects accessed by that transaction, the operations performed on those data objects, and the types of checks associated with those data objects (see

[1] An object identifier is a 64 bit value that consist of the id of the object manager that created the data object (16 bit) and a value that is the sum of the object manager's start-up timestamp and a counter (48 bit). Our object identifiers are simplified versions of universally unique identifiers (UUID) [Open97]. 16,000 ids per second can be generated by each object manager. More precisely, an object manager that has been running for *s* seconds and has already generated *n* ids after it has been started can immediately generate *s**16,000-*n* new ids.

Subsection 6.5.1). In the remainder of this subsection, we describe the database access operations performed during the commit phase of an FPT transaction.

After application code has called *commitTx()* to terminate a running FPT transaction, the transaction's push-down phase (see Subsection 5.7.4) starts and the FDO framework routes items to the appropriate data stores (see object routing described in Subsection 5.8.2). In the following commit phase, all data stores are accessed according to the items that were routed to them. An item is either a private version, which represents an insert, update, or delete operation, or a check to be performed on a data object's persistent state.

Let us assume that a set of items arrives at a given Oracle database. During the commit phase, our *OracleJDBCConnection* plug-in acts as follows:

1. The data objects to be processed are identified. For each of those data objects, there is either a private version or a check or both. For instance, for an employee data object with oid=4C0B724E, there might be a private version that reflects an update to the salary attribute and a version check for version number 42.
2. The data objects to be processed are sorted by their unique object identifier to ensure a defined lock order for the following SQL statements. This prevents local deadlocks (however, distributed deadlocks are still possible).
3. All data objects are processed, one after another. The actions for each data object are shown as pseudo-code in Table 6-5 and depend on the operation performed (insert/update/delete/read) and the types of check associated with the data object (version/existence/predicate/none). Thus, the table describes 16 combinations; five of them are invalid or do not apply.

   Example: Consider an FPT transaction that updated a given data object *D* in manual check mode and added a predicate check to *D*. During the commit phase, an SQL SELECT statement is executed to read the row that contains *D*'s persistent state. The row is looked up by its OID value. If no row is returned, then *D* must have been deleted and the current FPT transaction worked on stale data. In that case, the predicate check fails and the FPT transaction is aborted. However, if a row for *D* is returned, then the predicate is evaluated on that row. If the result of the evaluation is *true*, then the predicate check is successful and the row is updated with a dedicated SQL UPDATE statement. However, if the result of the evaluation is *false*, then the predicate check fails and the FPT transaction is aborted. Table 6-5 also provides details on fresh versions that are reported back to an object manager – this aspect is discussed in the end of this subsection.

	existence check	predicate check	version check	no check
insert	n/a (invalid combination)	n/a (invalid combination)	n/a (invalid combination)	INSERT new row, report new version
update	SELECT rows by oid 0 rows found: check failed, report deleted obj 1 row found: check ok, UPDATE row, report new version	SELECT rows by oid 0 rows found: check failed, report deleted obj 1 row found: eval predicate true: check ok, UPDATE row, report new version false: check failed, if newer version then report it	SELECT rows by oid 0 rows found: check failed, report deleted obj 1 row found: if current version then check ok, UPDATE row, report new version if newer version then report it, check failed	n/a (invalid combination)
delete	DELETE by oid 0 rows affected: check failed, report deleted obj 1 row affected: check ok, report deleted obj	DELETE by oid AND predicate 0 rows affected: check failed, SELECT by oid 0 rows found: report deleted obj 1 row found: report version found 1 row affected: check ok, report deleted obj	DELETE by oid AND version 0 rows affected: check failed, SELECT by oid 0 rows found: report deleted obj 1 row found: report newer version 1 row affected: check ok, report deleted obj	DELETE by oid, report deleted obj
read-only access	SELECT rows by oid 0 rows found: check failed, report deleted obj 1 row found: if current version then check ok if newer version then report it, check ok	SELECT rows by oid 0 rows found: check failed, report deleted obj 1 row found: eval predicate true: check ok false: check false if newer version then report it	SELECT rows by oid 0 rows found: check failed, report deleted obj 1 row found: if current version then check ok if newer version then report it, check failed	n/a (valid combination but no item is routed to data stores in that case)

Table 6-5. Actions performed in the commit phase of an FPT transaction (for each data object accessed within that transaction). SELECT, INSERT, UPDATE, and DELETE stand for the respective SQL statements.

UPDATE statements change only those attribute values that have actually been updated by an FPT transaction. In addition, an UPDATE statement always increments the version column, which stores the persistent counter of a data object. In the table, we do not distinguish between standard updates and delta writes (see Subsection 6.5.2). The values to write for delta writes are always calculated based on the row returned by the preceding SELECT statement.

All database access operations for a set of routed items are executed within a branch of a distributed transaction (see following subsection). Our implementation uses Oracle's default isolation level, which is *read committed*. All SELECT statements – except for those executed for deleted data objects – use the FOR UPDATE clause to pessimistically lock the accessed row until the transaction branch is terminated. Note that pessimistic locks are held for a short time only and occur only during the commit phase of an FPT transaction – the FDO framework's overall concurrency control mechanism is still optimistic.

After all database accesses have been performed, i.e., at the end of the commit phase of an FPT transaction, it is essential to propagate fresh public versions[1] of data objects back to the object manager that initiated the FPT transaction. This helps to keep the object manager's cache contents fresh. A fresh public version is reported in the following three cases:

- A new version of a data object has been created by the current FPT transaction (as a result of an insert, update, or delete statement). The new public version is reported because, in many cases, the corresponding data object will be accessed by following transactions working on the same object manager. Note that, in some cases, an object manager does not require to be explicitly informed of a new public version (e.g., for a newly created data object) because all information necessary for creating that version is already available. However, in other cases (e.g., when a data

[1] including information on deleted data objects

object is updated in combination with an existence or a predicate check), an object manager requires a fresh version from the database.

- A check failed, possibly because the FPT transaction had worked on stale data. The object manager is supplied with a fresh public version to improve the chance of success of future FPT transactions (which are likely to access the same data object(s), e.g., to retry the aborted transaction).
- An existence or predicate check has been performed successfully but the plug-in implementation detects that the FPT transaction has worked on stale data. The FPT transaction commits successfully because no version check has been requested. However, to improve the freshness of the object manager's cache, a fresh version is propagated.

Table 6-5 describes in detail when fresh versions are reported. Note that not only the root object manager of an FPT transaction but also each intermediate object manager that routed messages for that transaction benefits from the propagation of fresh versions.

6.7.3 Distributed Transactions

In the push-down phase of an FPT transaction, items are routed from the root object manager "down" the process topology to routing endpoints, i.e., object managers with direct access to appropriate data stores (JDBC access to Oracle databases in our case). During the commit phase, the object managers that are routing endpoints access the databases through the *OracleJDBCConnection* plug-in as described in the previous subsection. All database accesses on behalf of an FPT transaction are performed within a single distributed XA transaction (see Section 2.2 for XA transactions). By default, the root object manager (the object manager local to the initiator of the FPT transaction) plays the role of a transaction coordinator for the distributed transaction. This includes creation of a unique transaction identifier for the XA transaction and driving the two-phase commit protocol to terminate the transaction. An XA transaction consists of one or more transaction *branches*. In our implementation, each branch is uniquely identified by a path *P* of C1 connectors. The path starts at the root object manager of the FPT transaction to the routing endpoint object manager OM_i, to the database db_j.

All items that are routed together on the same path are processed within one transaction branch. It is possible that, within the same XA transaction, the same database is accessed by more than one routing endpoint object manager. Also, one object manager may access multiple databases. It is even possible that items are routed on different paths to the same object manager. In all such cases, different XA transaction branches are created because the path uniquely identifies a branch.

All transaction branches are executed concurrently. Each branch is initiated by a *PrepareRequestMessage* (see Section 6.3) and proceeds as follows:

1. The branch obtains an *XAConnection*[1] instance from a pool of available JDBC connections.
2. The branch calls *start* on the *XAResource* instance associated with the connection.
3. The connection is used to execute all SQL statements of the branch (see Subsection 6.7.2).
4. The branch calls *end* on the *XAResource* instance.
5. The branch calls *prepare* on the *XAResource* instance.
6. A *PrepareReplyMessage* containing a vote is propagated back to the transaction coordinator (following the reverse path of the *PrepareRequestMessage* that initiated the branch).

At this stage, the transaction coordinator collects all votes and, if no branch voted for a transaction abort, sends out *CommitRequestMessages*. However, if at least one branch voted for abort or after a timeout not

[1] *XAConnection* and *XAResource* are classes implemented by Oracle's JDBC driver.

all votes have been received, *RollbackRequestMessages* are sent out instead. One message per transaction branch is sent; the messages follow the same paths as used for *PrepareRequestMessages* and *PushDownRequestMessages* before. Finally, each transaction branch continues execution and is completed as follows:

7. The branch calls *commit* (or *rollback*) on the *XAResource* instance associated with the connection.
8. A *CommitReplyMessage* (or *RollbackReplyMessage*) containing a status value and fresh data object versions (see Subsection 6.7.2) is propagated back to the transaction coordinator.

We have implemented only those parts of the 2PC protocol that are essential for a proof-of-concept implementation. Restart and termination protocols, which provide fault tolerance and are important for production systems, have not been implemented.

Note that, with standard transaction identifiers for XA transactions (different branches of a transaction use the same global transaction id but different branch qualifiers), it would not be possible for two branches to concurrently access and prepare the same database. One way to solve that problem would be to wait until all branches have completed executing their SQL statements and then prepare each involved database exactly once (instead of executing *prepare* once per branch). However, that would also require an additional roundtrip of request/reply messages. In our implementation, we solve the problem in another way. The branch qualifier part of an XA transaction identifier (Xid) is not used and is always set to a default value. The global transaction id of an Xid is constructed by concatenating the unique FPT transaction identifier with a binary representation of the path, which uniquely identifies a branch. As a consequence, all Xids seem to belong to *different* XA transactions from the viewpoint of a database. That means that branches of the same distributed transaction that access the same database are executed by the database as if they belonged to different XA transactions. However, since our object routing mechanism guarantees that two branches of the same distributed transaction can never access the same data objects in the same database (i.e., there are no conflicts between different branches), the result of the execution is equivalent.

We enhanced our implementation with two optimizations:

- *Coordinator transfer.* By default, the root object manager becomes the transaction coordinator of an XA transaction. However, when the paths of all items to be routed in the push-down phase start with a common prefix, the task of coordinating the distributed transaction can be transferred together with the (single) *PushDownRequestMessage* that is sent by the root object manager. The receiving object manager may again delegate coordination to the next object manager on the path; that process is repeated until the path forks or a routing endpoint object manager is reached. Coordinator transfer is a well-known technique to optimize distributed commit processing and is, for example, described in [GR93].
- *One-phase commit.* When a distributed transaction consists of one branch only, a one-phase commit is used instead of a 2PC. The 8-step execution of a branch described above changes as follows: In Step 1, a standard connection instead of an *XAConnection* is taken from the pool. Steps 2, 4, and 7 are omitted. Step 5 is replaced with a *commit* call on the standard connection instance.[1]

Both optimizations improve performance. With coordinator transfer, the paths of messages exchanged in the 2PC protocol are shorter. With one-phase commit, database access is faster. However, there are more advantages. For example, when a client object manager is configured to have only a single server connection to another object manager, the client will never have to coordinate a distributed transaction. In

[1] Note that, after a one-phase commit, a *CommitRequestMessage* and a *CommitReplyMessage* (which correspond to the second phase of the 2PC protocol) are still exchanged. In principle, that round of messages could be eliminated. However, to avoid defining new or changing existing message types for the one-phase commit optimization (fresh versions are propagated via *CommitReplyMessages*, see Subsection 6.7.2), we kept the round of messages. Nevertheless, accessing a database within a non-distributed transaction is still a significant performance improvement compared to a two-phase commit.

all cases, the client can transfer coordination. This is important because, in most setups, it is not desirable to let (unreliable) client machines execute the 2PC protocol. Even when a client object manager is connected to multiple servers, client coordination can be avoided with an appropriate process topology and import/export scheme.

In many cases, the coordinator transfer optimization and the one-phase commit optimization can be combined. For example, consider an FPT transaction that consists of a single update access to a non-replicated data object. In that case, only a single path is used for routing and coordination can always be transferred to a routing endpoint object manager, which has direct access to the data object's home data store. In addition, the data object's persistent state can be updated with a fast, non-distributed transaction that is terminated by a one-phase commit.

6.8 Multi-Threading

An important property of enterprise applications – and thus of underlying enterprise application middleware – is performance. Note that, in practice, it is usually not the primary goal to minimize response time for an individual client request. Instead, it is typically much more important to achieve a high (average) transaction throughput so that the enterprise application can serve many concurrent clients while maintaining response time at an acceptable level.

With a single-threaded implementation and an object manager that sequentially processes transactions, only low transaction throughput could be achieved because only one processor would perform work on a multi-processor machine. Furthermore, either CPU work or IO could be performed by a single thread – but typically not both in parallel. For demonstrating the core features of our proof-of-concept implementation – custom and adaptable process topologies – a single-threaded implementation would have been sufficient. However, for a thorough evaluation (including transaction throughput and transaction conflicts – see Chapter 7), a single-threaded implementation would not yield meaningful results. For that reason, we designed our FDO framework to be fully multi-threaded. Each part of the framework has been optimized for high transaction throughput. This includes that threads never spend their time *busy waiting*, i.e., threads never consume CPU cycles while waiting for an event.

6.8.1 Multi-Threaded Communication Plug-ins

In the FDO framework, protocol adapter plug-ins (which are responsible for inter object manager communication or database access, see Figure 6-1) define their own threading model. Each plug-in instance uses one or more dedicated private threads to perform its work. When a thread of the core of an object manager wants to send a message to a remote site, it identifies an appropriate plug-in instance and inserts the message into the message queue maintained by the plug-in instance. The plug-in instance's private threads are then responsible for asynchronously marshalling and sending the message, while the object manager core thread immediately continues processing. A message from a remote object manager received by one of the plug-in instances is handled accordingly – a plug-in thread that reads a message from the network first unmarshals it, then inserts it into a queue maintained by the core object manager, and finally continues processing. The message is then asynchronously processed by an object manager core thread. The types of messages internally passed between the object manager core and plug-in instances are the same as shown in Table 6-1. As described in Section 6.3, our *SocketConnection* plug-in employs two private threads per connection; one for reading data from a socket and one for writing reading data to a socket. In contrast to that, the *NIOSocketConnection* plug-in employs only one thread per plug-in instance/port.

6.8.2 Multi-Threaded Database Access

From the viewpoint of the object manager core, the *OracleJDBCConnection* plug-in employed for database access (see Section 6.7) behaves like other plug-ins. For example, a *QueryRequestMessage* can be handed over to a *SocketConnection* plug-in so that the message is sent to an underlying object manager, but it can also be handed over to an *OracleJDBCConnection* plug-in so that the query is executed directly on an underlying Oracle database. The *OracleJDBCConnection* plug-in maintains a pool of private worker threads which work on database connections they obtain from a connection pool. When an object manager hands over a request message to an *OracleJDBCConnection* plug-in, a worker thread of the plug-in first processes the message by accessing the underlying Oracle database (see Subsection 6.7.2), then constructs an appropriate reply message from the result (e.g., a *QueryReplyMessage*), and finally hands over the reply message to the object manager core.

By default, 20 worker threads and 50 pooled JDBC connections are used by an instance of the *OracleJDBCConnection* plug-in. Although each running worker thread works on at most one connection at any given time, there are more connections than threads because some connections can be in the *reserved* state. When the plug-in processes a *PrepareRequestMessage*, an XA *prepare* call is performed and the current JDBC connection is marked as reserved. The connection remains reserved until an appropriate *CommitRequestMessage* is received by the plug-in and an XA *commit* or *rollback* call terminates the current distributed transaction (see steps 5 and 7 in Subsection 6.7.3). To avoid situations where too many or all connections are simultaneously in the *reserved* state (which may reduce throughput and could lead to distributed deadlocks), the *OracleJDBCConnection* plug-in uses a priority queue where messages are buffered before they are processed. When multiple messages are waiting in the queue, priority is always given to *CommitRequestMessages* and *RollbackRequestMessages* because processing those types of messages frees reserved connections. All other message types (*PushDownRequestMessage*, *PrepareRequestMessage*, and *QueryRequestMessage*) are assigned a lower priority. In addition, when the number of reserved connections exceeds a given threshold, all further *PrepareRequestMessages*[1] remain in the queue until connections are freed again.

6.8.3 Multi-Threaded Object Manager Core

The core of an object manager uses a separate pool of worker threads and thus is decoupled from plug-in instances. By default, two worker threads are used for each object manager that is a pure client (i.e., its configuration does not define any client listeners). 15 worker threads are used for each server object manager. When a plug-in hands over a request message to the object manager core, a corresponding *Task* instance is created and queued – for instance, a *QueryTask* instance for a *QueryRequestMessage* or a *PrepareTask* for a *PrepareRequestMessage*. A task encapsulates the state and context of a request. A task is either

- *running* (currently being executed by a worker thread),
- *ready* (waiting for a worker thread to execute the task),
- *blocked* (the task has sent one or more request messages and is waiting for all reply messages), or
- *completed.*

Internally, all tasks are organized as state machines. For example, the states of a *QueryTask* are *initial*, *waiting_for_replies*, and *success*. In addition, there are several error states. A state transition may occur when either (1) a task is *ready*, (2) all reply messages expected by a *blocked* task have arrived, or (3) a *blocked* task experiences a timeout. The state transition is executed when a task's *doNextStep* method is executed by a worker thread. For instance, a *QueryTask* might be created and processed as follows:

[1] *PrepareRequestMessages* that require a *one-phase commit* only (see optimizations in Subsection 6.7.3) are processed normally because they do not reserve connections.

1. A plug-in receives a *QueryRequestMessage* from a client object manager and hands over the message to the object manager core, where a *QueryTask* instance is created. In the beginning, the new task is *ready* and in the *initial* state.
2. Eventually, a worker thread calls the *doNextStep* method of the task. The task (now *running*) analyzes the given query, performs a query routing, and (in this particular example) sends out three *QueryRequestMessages* by handing them over to appropriate plug-ins (for communication or database access). The task registers an interest in all three reply messages it expects, moves to the *waiting_for_replies* state and becomes *blocked*.
3. Eventually, reply messages arrive and plug-ins hand them over to the object manager core. The first two (of the three) *QueryReplyMessages* the *QueryTask* has registered an interest for are buffered. When the third message is handed over, the task becomes *ready* again.
4. Eventually, a worker thread (not necessarily the one from Step 2) calls the *doNextStep* method of the task. The task (now *running*) reads all three reply messages, integrates the partial query results contained in the reply messages, constructs a *QueryReplyMessage*, and hands over the message to a plug-in so that the query result is (asynchronously) sent to the client object manager mentioned in Step 1. Finally, the task moves to the *success* state, becomes *completed* and can be garbage collected.

The same mechanism as described above is used when a task is initiated by local application code (e.g., an application thread calls the object manager's query method, see Listing 6-1) instead of a remote object manager. However, the initial request message and the final reply message are omitted in that case. For associating incoming reply messages with waiting threads (see Step 3 above), we employed a modified version of the *reply manager* design pattern we proposed in [Hart99].

With our approach to multi-threading, IO is strictly decoupled from processing in the object manager's core. Also, different plug-in instances and their IO are isolated from each other. All activities can run in parallel. In addition, an object manager can cope well with overload situations because messages and tasks that cannot be processed immediately are buffered and then processed asynchronously. Except for the queue of the *OracleJDBCConnection* plug-in, all internal queues are FIFO queues. However, it would be straightforward to replace them with priority queues so that an overloaded system can give higher priority to certain message types or clients.

Another advantage of our multi-threaded design is that, with tasks designed as state machines, the number of concurrent tasks is completely decoupled from the number of worker threads. Since *blocked* tasks do not require that a thread is associated with them, a constant number of threads can be used to handle an arbitrary number of concurrent tasks. In contrast to that, approaches based on synchronous remote invocations (e.g., with RPCs or object middleware), typically require one thread per request. Once a request is dispatched to a thread, the thread remains assigned to the request (and, if necessary, is blocked on IO) until the request is completed. To avoid creating too many threads, requests could be queued before they are processed by threads, but this way it is not possible to partially process many requests. All in all, our state machine based approach requires fewer threads because worker threads are only blocked when there is no work to do.

Note that, on server machines with many processors, it might not be the best approach to assign all processors to a single process. In many cases, better performance can be expected when multiple multi-threaded processes (each with a separate object manager) are run on a server machine to reduce synchronization overhead.

6.9 Object and Query Routing

In Section 5.8, we described object routing and query routing at an architectural level. We defined several constraints for selecting target data stores and valid paths to those data stores. Moreover, we explained that a simple approach to routing (i.e., selecting target data stores and paths first and then propagate items to be routed) conflicts with the limited visibility requirement and makes optimizations difficult. In this section, we outline the particular routing implementation of our FDO framework, which avoids the two problems mentioned above. We continue to use the example routing that has been described in Subsection 5.8.2. In the following, we focus on routing a single item– routing of multiple items is described in Subsection 6.9.4.

6.9.1 Basic Approach

The basic idea is that an object manager calculates a routing *incrementally*. Decisions regarding target data stores and paths are deferred for as long as possible and made during the propagation process. Each object manager that receives a set of items to be routed (within a *QueryRequestMessage* or a *PushDownRequestMessage*) recursively delegates routing decisions to its server object managers – unless RI-tree, import/export schema, and process topology require a local decision.

For instance, consider the routing shown in Figure 5-10. In the example, a single item (a private version of a newly created data object) is routed from a root object manager to target data stores, into which the new data will be inserted. The data object belongs to a domain for which the RI-tree given in Figure 5-3 has been defined. Initially, the root object manager has three options for selecting target data stores (see Subsection 5.2.3): (1) ds_2, ds_3, and ds_5, (2) ds_2, ds_3, and ds_6, or (3) ds_8, and ds_9. In principle, the item can be routed from the root object manager to *P1*, *P3*, or *P4* (or to any combination of these three) in the first step. Let us assume that the root object manager routes the item to *P1* as shown in the figure. This decision implicitly means that only options (1) and (2) remain valid choices for home data stores since neither ds_8 nor ds_9 can be accessed via *P1*. However, there is no need for the root object manager to decide whether option (1) or (2) should be selected. Instead, that decision can be delegated to *P1* – together with the *PushDownRequestMessage* that propagates the item. *P1* in turn delegates the decision to *P3*, which finally decides whether ds_5 or ds_6 is included in the set of home data stores. Simultaneously, *P1* routes the item to *P2*, since it is clear at that stage that both ds_2 and ds_3 must be included in the set of home data stores.

In our implementation, each object manager uses a *local routing function* to make local routing decisions (see Figure 6-5). The local routing function is implemented as part of the router component depicted in Figure 6-1. Together with each item, a set of possible target data stores *targetCandidatesIn* is routed. The set and the item to be routed are input parameters of the local routing function. The task of the function is to decide to which directly connected servers (server object managers or data stores) an item should be routed and to produce a set of possible target data stores *targetCandidatesOut* for each of those servers. Underlying server object managers then use the produced *targetCandidatesOut* set as an input parameter of their local routing function.

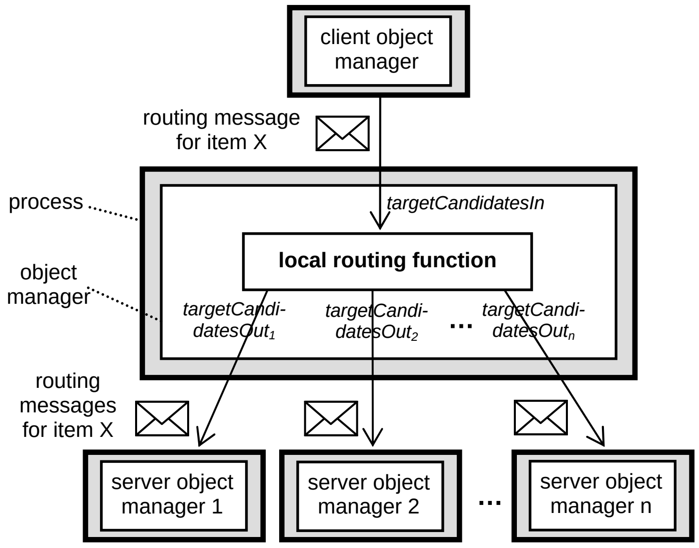


Figure 6-5. Each object manager performs a local routing using a local routing function.

Figure 6-6 shows the example routing from Figure 5-10, annotated with sets of target data store candidates for each process involved in the routing. In the root object manager, which initiates the routing, *targetCandidatesIn* is set to *{ds$_2$, ds$_3$, ds$_5$, ds$_6$, ds$_8$, ds$_9$}*, which are all the potential home data stores (all leaf nodes of the new data object's RI-tree that are accessible). Each C1 connector is annotated with the *targetCandidatesOut* value produced by the sending object manager's local routing function.

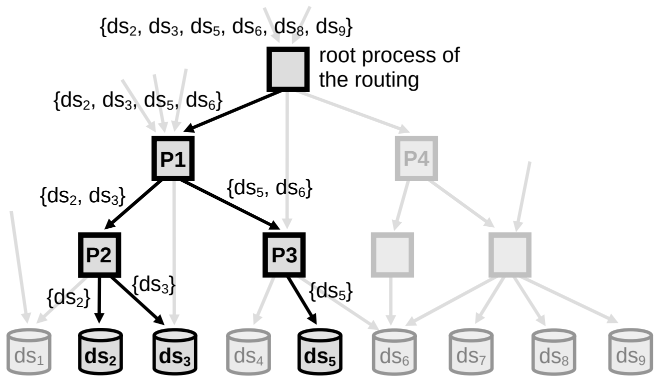


Figure 6-6. Example routing of a private version, annotated with sets of target data store candidates.

An alternative routing, which is also correct, is shown in Figure 6-7. In this example, the root object manager decides to route the item to *P1* and also to *P4*. However, that decision also forces the root object manager to immediately select *ds$_2$*, *ds$_3$*, and *ds$_6$* as target data stores. Note that, unlike the routing shown in Figure 6-6, the decision whether *ds$_5$* or *ds$_6$* should be selected cannot be delegated here because *P4* has only access to *ds$_6$* (via *P5*). *P1* could access *ds$_5$*, but if the root object manager included *ds$_5$* also in the *targetCandidatesOut* for *P1*, then the item would be routed to both *ds$_5$* and *ds$_6$*, which is not compliant with the RI-tree. While the root object manager selects all target data stores in this example, it does not determine the paths to target data stores and delegates that decision to *P1* and *P4*.

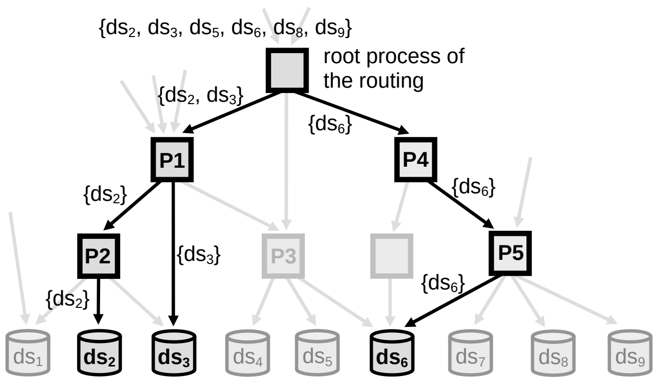


Figure 6-7. An alternative routing.

6.9.2 Initial Parameters and Routing Matrices

All object managers involved in the routing (except for the root object manager) use the *targetCandidatesOut* parameter supplied by a client object manager as *targetCandidatesIn* parameter for their local routing function. The root object manager sets the initial *targetCandidatesIn* value depending on the type of item to be routed (see Subsection 5.8.3 for a description of the five different types of items). In the following, we use definitions from sections 5.2.3 and 5.8.3.

Let *dom* be the domain associated with the item to be routed and *RI* the RI-tree defined for *dom*. Furthermore, we define *accessibleDataStores* := *{ds$_i$* | the root object manager imports *(dom, ds$_i$)* from at least one server*}*.

- For an item that is a set query, the root object manager sets *targetCandidatesIn* := *accessibleDataStores*. If there is no $G \in$ *queryOptions(RI)* such that $G \subseteq$ *targetCandidatesIn*, then a routing error occurs.
- For an instance query for data object *D* or a check (optimistic lock) on *D*, the root object manager sets *targetCandidatesIn* := *accessibleDataStores* $\cap$ *home(D)*. If *targetCandidatesIn* is empty, then a routing error occurs.
- For a private version of a newly created data object, the root object manager sets *targetCandidatesIn* := *accessibleDataStores*. If there is no $G \in$ *insertOptions(RI)* such that $G \subseteq$ *targetCandidatesIn*, then a routing error occurs.
- For a private version of an existing data object *D*, the root object manager sets *targetCandidatesIn* := *accessibleDataStores* $\cap$ *home(D)*. If *targetCandidatesIn* $\neq$ *home(D)*, then a routing error occurs.

A routing error is the result of an inappropriate configuration that does not give an object manager sufficient imports to successfully initiate a routing. For example, if the root object manager shown in Figure 6-7 had only access to *P3*, but not to *P1* or *P4*, then it could not produce a valid local routing for a private version of a newly created data object. In case of a routing error, the corresponding FPT transaction is aborted.

Internally, a local routing function first creates an *input routing matrix* from the *targetCandidatesIn* input parameter. Then the function calculates a local routing. The result is represented by an *output routing matrix*. Finally, the output routing matrix is transformed into one or more *targetCandidatesOut* sets.

A routing matrix *R* is a matrix that contains a “1” or a “0” value for each combination of a (possible) target data store *i* and a directly connected server *k*. In an input routing matrix, R_{ik}=1 means that it is allowed to route an item via server *k* to data store *i*; R_{ik}=0 means that it is forbidden. In an output routing matrix, each R_{ik} that is set to “1” means that the item is to be routed to server *k* and that data store *i* is included in *targetCandidatesOut$_k$*. An input routing matrix *IRM* is constructed as follows: $IRM_{ik}=1 \Leftrightarrow$ *(dom, ds$_i$)* is imported by the local object manager from server *k* and *ds$_i$∈targetCandidatesIn*.

Figure 6-8 shows input and output routing matrices for the example routings depicted in figures 6-6 and 6-7. The matrices shown are those used and produced by the local routing function of the root object manager. To make the matrix representation more readable, rows and columns that consist entirely of “0” values in all three matrices are omitted.

targetCandidatesIn for the root object manager: {ds$_2$, ds$_3$, ds$_5$, ds$_6$, ds$_8$, ds$_9$}

input routing matrix:

	P1	P3	P4
ds$_2$	1	0	0
ds$_3$	1	0	0
ds$_5$	1	1	0
ds$_6$	1	1	1
ds$_8$	0	0	1
ds$_9$	0	0	1

output routing matrix (a):

	P1	P3	P4
ds$_2$	1	0	0
ds$_3$	1	0	0
ds$_5$	1	0	0
ds$_6$	1	0	0
ds$_8$	0	0	0
ds$_9$	0	0	0

=> targetCandidatesOut for P1: {ds$_2$, ds$_3$, ds$_5$, ds$_6$}

output routing matrix (b):

	P1	P3	P4
ds$_2$	1	0	0
ds$_3$	1	0	0
ds$_5$	0	0	0
ds$_6$	0	0	1
ds$_8$	0	0	0
ds$_9$	0	0	0

=> targetCandidatesOut for P1: {ds$_2$, ds$_3$}, targetCandidatesOut for P4: {ds$_6$}

Figure 6-8. Input and output matrixes used by the root object manager of the example routings shown in figures 6-6 (a) and 6-7 (b).

6.9.3 Constraints

The main task of a local routing function is to produce an output routing matrix for a given input routing matrix. Often, there are several different possibilities of routing an item. However, not all possible output routing matrices necessarily correspond to valid routings, i.e., routings that adhere to the data distribution scheme. In Subsection 5.8.3, we defined rules for selecting valid paths and target data stores. In this subsection, we describe constraints that, when observed by a local routing function, guarantee that an output routing matrix always complies with the rules given in Subsection 5.8.3.

As a prerequisite, we define four functions. All functions work on the RI-tree defined for the domain that is associated with the item to be routed.

iNeighbors(x) := {y | the lowest common ancestor of ds$_x$ and ds$_y$ is an I-node}

rNeighbors(x) := {y | the lowest common ancestor of ds$_x$ and ds$_y$ is an R-node}

iNeighborGroups(x) returns a set of sets *S*, which is constructed as follows:

```
start with $S := \varnothing$
for each ancestor $n$ of $x$ that is an I-node do
    for each direct descendant $m$ of $n$ that is neither $x$ nor an ancestor of $x$ do
        add the set of all leaf nodes that are descendants of $m$ (or $\{m\}$, if $m$ is a leaf node) to $S$
```

rNeighborGroups(x) returns a set of sets *S*, which is constructed as follows:

```
start with $S := \varnothing$
for each ancestor $n$ of $x$ that is an R-node do
    for each direct descendant $m$ of $n$ that is neither $x$ nor an ancestor of $x$ do
        add the set of all leaf nodes that are descendants of $m$ (or $\{m\}$, if $m$ is a leaf node) to $S$
```

Examples for the RI-tree shown in Figure 5-3 are given below:

iNeighbors(ds$_2$) = {ds$_8$, ds$_9$}
iNeighbors(ds$_3$) = {ds$_8$, ds$_9$}

iNeighbors(ds_5) = {ds_6, ds_8, ds_9}
rNeighbors(ds_5) = {ds_2, ds_3}
iNeighborGroups(ds_5) = {{ds_6},{ds_8, ds_9}}
rNeighborGroups(ds_9) = {{ds_9}}

In Subsection 5.8.3, we listed five different types of items that can be routed. Depending on the type of item, the following constraints apply for producing an output routing matrix *ORM* from a given input routing matrix *IRM*.

Constraints for all types of items:

(a) *ORM* contains at least one “1” value

(b) $IRM_{ik} = 0 \Rightarrow ORM_{ik} = 0$

(c) each row in *ORM* contains at most one “1” value

Additional constraints for set query items:

(d) $ORM_{ik} = 1 \Rightarrow$ all ORM_{pq} with $p \in rNeighbors(i) \wedge q \neq k$ have to be “0”

(e) $ORM_{ik} = 1 \Rightarrow$ for each $G \in iNeighborGroups(i)$, there is at least one $g \in G$ such that row g of *ORM* contains a “1” value

Additional constraint for instance queries and checks (optimistic locks):

(f) exactly one column of *ORM* contains “1” values; all other columns consist entirely of “0” values

Additional constraints for private versions of newly created data objects:

(g) $ORM_{ik} = 1 \Rightarrow$ all ORM_{pq} with $p \in iNeighbors(i) \wedge q \neq k$ have to be “0”

(h) $ORM_{ik} = 1 \Rightarrow$ for each $G \in rNeighborGroups(i)$, there is at least one $g \in G$ such that row g of *ORM* contains a “1” value

Additional constraint for private versions of existing data objects:

(i) for each row *i* in *IRM* that contains at least one “1” value, row *i* in *ORM* must contain a “1” value

Constraint (a) guarantees that each item is routed to at least one server. Constraint (b) ensures that each item is routed in compliance with the import/export scheme. Constraint (c) ensures that, when each item is routed to multiple servers, the *targetCandidatesOut* sets produced for those servers are pairwise disjoint. Without that constraint, an item could be routed on different paths to the same data store in a topology that contains meshes. With constraint (d), we prevent the situation that a set query is performed redundantly on replicated data. Instead, we ensure that each query item is routed to such a set of data stores that all partial query results are disjoint. Constraint (e) guarantees that all result data objects are found, i.e., each query item is routed to such a set of data stores that the union of all partial query results represents the complete query result. Constraint (f) ensures that each item is routed to exactly one server. Multiple paths are not permitted because an instance query or check needs only be routed to one data store. Constraints (g) and (h) guarantee that a newly created data object is inserted into a set of data stores in compliance with the data distribution scheme, i.e., a valid insert option is selected. Finally, constraint (i) ensures that an update or delete is routed to all home data stores of a data object (ROWA replication).

6.9.4 Routing Function Implementation

Depending on the process topology, data distribution scheme, and import/export scheme, there can be different possibilities of routing an item. For instance, in Subsection 6.9.1, we presented an example and two different ways to route an item. A local routing function has to select one routing according to some criteria. The local routing function implemented in our FDO framework simply chooses a random routing to achieve basic load distribution among processes and data stores.

The routing function calculates a routing matrix from a given input routing matrix. A recursive function that calculates one row of the output matrix per step is used. With backtracking, a routing is calculated that satisfies the constraints described in the previous subsection. Before the output matrix is calculated, columns and rows are randomly permuted (logically) so that the first valid output matrix found represents a random routing. When multiple items are to be routed, the routing function processes one item after another. Items that are routed to the same server – i.e., items with paths that have a common prefix – are propagated together within a single message (coarse-grained communication). Typically, routing matrices are relatively small (e.g., one to twenty cells) and thus can be processed quickly. To prevent routing from becoming a bottleneck when large amounts of items are routed, we implemented the following optimizations:

- *Early pruning.* Constraints are checked early and limit choices for other rows so that backtracking is reduced significantly.
- *Bit sets.* All matrix operations are implemented using bit sets, which dramatically improves performance.
- *Chain routing.* Before all items in a given set are routed, the set is first sorted (according to item type, domain, and *targetCandidatesIn*). Then all items are routed, one after another. However, before an item is passed to the routing function, it is first tested whether the item can be routed exactly like its predecessor. This test is executed extremely efficiently. If the test is positive, the routing of the predecessor is used for the current item, too; otherwise the routing function is executed. With chain routing, large sets of items can be routed with typically just a few calls to the routing function.

Because of the optimizations outlined above, routing did not play any significant role in the benchmarks presented in Chapter 7.

Note that our routing function implementation is just one particular solution. There are many other possible routing functions with more sophisticated optimization criteria that could be implemented. For example, an object manager could track the response times or request queue lengths of servers to realize a load balancing. Or, certain data objects could be clustered together in data stores. The bandwidth of connections, the fill ratio of data stores, or the priority of clients/requests could also be taken into account. Our random routing function provides only basic load distribution, but it demonstrates our incremental approach to routing well. In particular, the limited visibility requirement is met. Furthermore, the autonomy of each object manager in the process topology is preserved. Each object manager can optimize locally and delegates items and routing decisions to server object managers. As long as the constraints defined in Subsection 6.9.3 are satisfied, it is even possible that each object manager employs a different routing function.

6.10 Discussion

In this section, we discuss selected aspects in more detail.

6.10.1 Programming Language

Our proof-of-concept implementation heavily relies on Java. For example, we used weak and soft references, garbage collection, class libraries for low-level socket communication, Java serialization, and JDBC. However, in principle, an object manager can be implemented in any programming language, for instance in C++. Conceptually, nothing changes with another programming language. However, for communication between object managers implemented in different programming languages, communication plug-ins that do not rely on Java serialization have to be used.

6.10.2 Object Managers in Thin Clients

In terms of code size, our object manager implementation is a relatively lightweight component. Packaged as a Java archive, an object manager requires less than 200 kilobytes. The small size makes it possible for a client application, including its object manager component, to be deployed as a Java applet. A Java applet can be run by any thin client that is equipped with a recent web browser. With an appropriate process topology, it can be prevented that such an applet ever has to coordinate a distributed transaction (see Subsection 6.7.3). An alternative to applets is Java Web Start (bundled with Sun's Java 2 Platform, Standard Edition), which allows clients to automatically download, install, and update Java client applications.

6.10.3 Heterogeneous Topologies

Our FPT architecture for enterprise applications represents a homogeneous approach to middleware – each process of a process topology has to be equipped with an object manager component. However, in many cases, the FPT architecture and our implementation can easily be integrated with conventional processes/applications that do not use object managers:

- *Access to conventional processes.* Often, transactional data is managed by conventional processes, for instance, Enterprise JavaBeans application server processes, CORBA servers, or applications based on object/relational mapping frameworks. With an appropriate plug-in for data access, conventional processes can be treated like transactional data stores. For example, a plug-in for accessing data managed by EJB servers could translate FDO request messages into appropriate sequences of RMI invocations on EJB entity beans. Ideally, a conventional process exposes a sufficiently rich interface (e.g., for distributed transaction management) so that the full range of features of the FDO framework can be used. When a conventional process provides necessary functionality but does not expose it, it might be possible to place a plug-in in that process. Such a plug-in could be a server-side adapter for FPT-based processes.
- *Conventional processes as clients.* Also, conventional processes can access data objects managed by object managers of FPT-based processes. Here, we have two options:

  The first option is that *application code* in an FPT-based process acts as a facade and provides services to conventional clients (which, in turn, could serve other conventional processes). For communication, any communication protocol can be used. As an example, we implemented an RMI-based approach, where application code wraps FDO data objects and exposes them as RMI objects to conventional clients. That example is studied in Section 7.4 and depicted in Figure 7-9. Another example is application code that implements a Java servlet that dynamically generates HTML documents from transactional data objects managed by the local object manager. The HTML documents are then delivered to conventional (ultra-thin) clients via HTTP.

  The second option is to let client processes directly communicate with object managers of FPT-based processes. A client does not need a local object manager for communication – in principle, it is sufficient to directly send and receive FDO request/reply messages to perform basic operations on data objects. To simplify connectivity for conventional clients, an object manager could use, for instance, an XML-based communication plug-in.

6.11 Summary

In this chapter, we presented our proof-of-concept implementation. The implementation is a middleware framework that realizes all concepts of the FPT architecture. In particular, arbitrary process topologies can be conveniently defined through configuration and adapted through reconfiguration. We outlined the overall design of the framework and provided various implementation details regarding configuration, remote communication, concurrency control, caching, database access, XA transaction management, multi-threading, and routing.

Note that, in many respects, our framework goes beyond a simple proof-of-concept implementation. We implemented a broad range of features and tools that address practical requirements of many enterprise applications and make development of such applications more convenient. In addition, we implemented several optimizations to achieve high transaction throughput. The most important performance optimizations are summarized below:

- *Multi-threading.* Our object manager implementation is fully multi-threaded and utilizes multiple processors of a machine, if available. With asynchronous processing and our state machine based approach to tasks, an object manager can process more concurrent tasks using fewer threads. For concurrent database access, JDBC connection pooling is employed.
- *Advanced concurrency control features.* In addition to standard concurrency control, our framework offers several advanced concurrency control features (such as field calls) that allow application developers to fine-tune consistency and performance on a per-object basis. This is especially useful for dealing with hotspot data as we show in Section 7.7.
- *Commit protocol optimizations.* With coordinator transfer, the paths of messages exchanged in the 2PC protocol are made shorter. In addition, when only one database is accessed, performance is improved by using a one-phase commit instead of a two-phase commit.
- *Transaction logging with bundled force writes.* Although recovery is not fully implemented, object managers write records to a transaction log while coordinating XA transactions. We implemented a transaction log in order to obtain realistic performance results for the scenarios in Chapter 7. To prevent the transaction log from becoming a bottleneck, we implemented an optimization that groups multiple log writes (asynchronous and/or forced writes) into a single disk access when load is high.
- *Load distribution.* For object and query routing, we use a local routing function based on a random scheme. Thereby, we achieve basic load distribution among processes and data stores.
- *Chain routing.* When sets of items are routed (because an FPT transaction accessed multiple data objects), our implementation routes subsets of items together, if possible. Thereby, fewer calls to the local routing function are necessary.
- *Join queries.* Our framework provides limited support for join queries. Join queries help to avoid a fine-grained communication style that would occur if join functionality were realized by application code through standard queries and navigational access.
- *Scalable IO*. We implemented a communication plug-in that uses advanced socket-based communication using Java’s new NIO package, which provides low-level access to socket management and scales much better to many active client connections.