BEA Tuxedo®

CORBA Technical Articles

Version 10.0
Document Released: September 28, 2007

Contents

4. Client Data Caching Design Pattern

CHAPTER 1

The CORBA Programming Model

CORBA is a specification for creating distributed object-based applications. The CORBA architecture and specification were developed by the Object Management Group (OMG). The OMG is a consortium of several hundred information systems vendors. The goal of CORBA is to promote an object-oriented approach to building and integrating distributed software applications.

The CORBA specification provides a broad and consistent model for building distributed applications by defining:

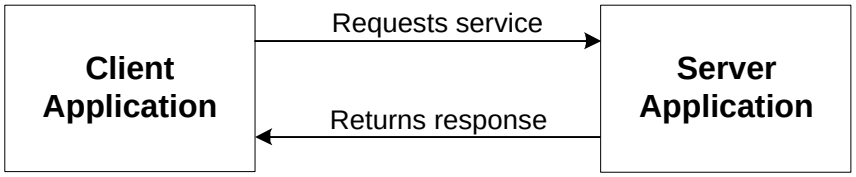
- An object model for building distributed applications.
- A common set of application programming objects to be used by the client and server applications.
- A syntax for describing the interfaces of objects used in the development of distributed applications.
- Support for use by applications written in multiple programming languages.

The CORBA specification describes how to develop an implementation of CORBA. It also describes programming language bindings that developers use to develop applications.

To illustrate the advantages of using the CORBA architecture, this section compares early client/server application development techniques to CORBA development techniques.

Pre-CORBA Approach to Client/Server Development

Client/server computing is an application development methodology that allows programmers to distribute processing among networked machine systems, thus enabling more efficient use of machine resources. In client/server computing, an application consists of two parts: the client application and the server application. These two applications typically run on different machines, connected by a network, as shown in the following figure.



The client application makes requests for information or services and typically provides users with a means to display results. The server application fulfills the requests of one or more client applications and usually performs compute-intensive functions.

The key advantages of the client/server model are:

- Computing functions run on the most appropriate machine system.
- Developers can balance the load of application processing among several servers.
- Server applications can be shared among numerous client applications.

For example, desktop systems provide many business users with an easy-to-use graphical environment for displaying information. However, desktop systems may have restricted disk space and memory and are typically single-user systems. Larger, more powerful machine systems are better suited to perform compute-intensive functions and provide multiple user access and shared database access.

Therefore, larger systems usually run the server portion of the application. In this way, distributed desktop systems and networked servers provide a perfect computing environment for deploying distributed client/server applications.

Although the non-CORBA client/server approach provides the means to distribute processing in a heterogeneous network, it has the following disadvantages:

- For communications, the client application must know how to access the server application, including any necessary network protocol information.

Client/server applications might use the same, single network protocol or different protocols. If they use multiple protocols, the applications must logically repeat the protocol-specific code for each network.

- Applications must handle data format conversions when they are integrated with machines that use different data formats.

  For example, some machines read an integer value from the lowest byte address to the highest (little endian), while others read the highest byte address to the lowest (big endian). Some machine systems might also use different formats for floating-point numbers or text strings. If an application sends data to a machine that uses a different data format, but the application does not convert the data, the data is misinterpreted.

  Transporting data over the network and converting it to its proper representation on the target system is called data marshaling**.** In many non-CORBA client/server models, applications must perform all data marshaling. Data marshaling requires that the application use features of the network and operating system to move data from one machine to another. It also requires that the application perform all data format translations to ensure that the data is read in the same way it was sent.

- There is less flexibility for extension of the application.

  The non-CORBA client/server approach ties the client and server applications together. Therefore, if either the client or server application changes, the programmer must change the interface, network address, and network transport. Additionally, if the programmer ports the client and server applications to a machine that supports a different network interface, the programmer must create a new network interface for those applications.

CORBA Approach to Client/Server Development

The CORBA model provides a more flexible approach to developing distributed applications. The CORBA model:

- Formally separates the client and server portions of the application

  A CORBA client application knows only how to ask for something to be done, and a CORBA server application knows only how to accomplish a task that a client application has requested it to do. Because of this separation, developers can change the way a server accomplishes a task without affecting how the client application asks for the server application to accomplish the task.

- Logically separates an application into objects that can perform certain tasks, called operations

CORBA is based on the distributed object computing model, which combines the concepts of distributed computing (client and server) and object-oriented computing (based on objects and operations).

In object-oriented computing, objects are the entities that make up the application, and operations are the tasks that a server can perform on those objects. For example, a banking application could have objects for customer accounts, and operations for depositing, withdrawing, and viewing the balance in the accounts.

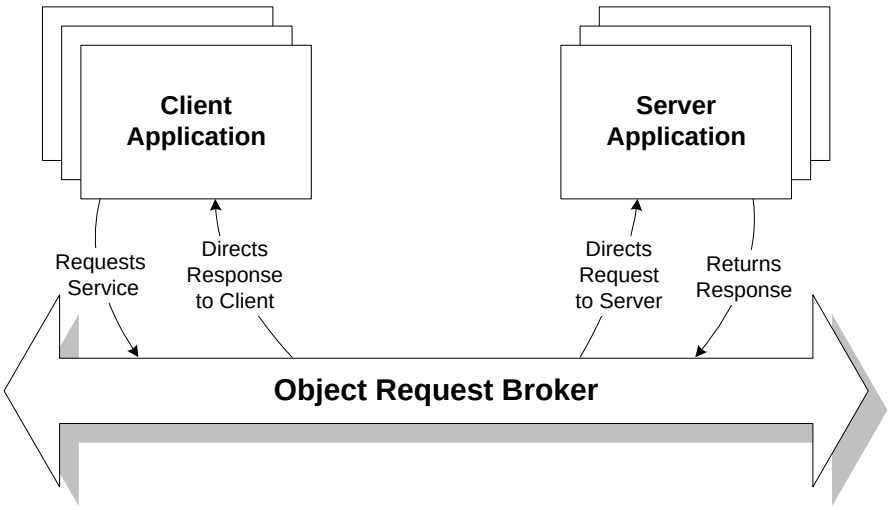
- Provides data marshaling to send and receive data with remote or local machine applications

  For example, the CORBA model automatically formats for big or little endian as needed. (Refer to the preceding section for a description of data marshaling.)

- Hides network protocol interfaces from the applications

  The CORBA model handles all network interfaces. The applications see only objects. The applications can run on different machines and, because all the network interface code is handled by the ORB, the application does not require any network-related changes if it is later deployed on a machine that supports a different network protocol.

The CORBA model allows client applications to make requests to server applications, and to receive responses from them without direct knowledge of the information source or its location. In a CORBA environment, applications do not need to include network and operating system information to communicate; instead, client and server applications communicate with the Object Request Broker (ORB). The following figure shows the ORB in a client/server environment.



CORBA defines the ORB as an intermediary between client and server applications. The ORB delivers client requests to the appropriate server applications and returns the server responses to the requesting client application. Using an ORB, a client application can request a service without knowing the location of the server application or how the server application will fulfill the request.

In the CORBA model, client applications need to know only what requests they can make and how to make the requests; they do not need to be coded with any implementation details of the server or of the data formats. Server applications need only know how to fulfill the requests, not how to return data to the client application.

This means that programmers can change the way a server application accomplishes a task without affecting how the client application asks for the server application to accomplish that task. For example, as long as the interfaces between the client and the server applications do not change, programmers can evolve and create new implementations of a server application without changing the client application; in addition, they can create new client applications without changing the server applications.

CHAPTER 2

CORBA Objects

Before any discussion of CORBA programming can be meaningful, it is important to have a clear understanding of what a CORBA object is and the object terminology used throughout the BEA Tuxedo information set.

This topic includes the following sections:

- Definition of a CORBA Object
- How a CORBA Object Comes into Being
- Components of a CORBA Object
- Where an Object Gets Its Operations
- How Object Invocation Works

There are a number of variations on the definition of an object, depending on what architecture or programming language is involved. For example, the concept of a C++ object is significantly different from the concept of a CORBA object. Also, the notion of a Component Object Model (COM) object is quite different from the notion of a CORBA object.

Most importantly, the notion of a CORBA object in this chapter is consistent with the definition presented by the Object Management Group (OMG). The OMG has a number of specifications and other documents that go into complete details on objects.

Definition of a CORBA Object

A CORBA object is a virtual entity in the sense that it does not exist on its own, but rather is brought to life when, using the reference to that CORBA object, the client application requests an operation on that object. The reference to the CORBA object is called an **object reference**. The object reference is the only means by which a CORBA object can be addressed and manipulated in an BEA Tuxedo system. For more information about object references, see *Creating CORBA Server Applications* in the BEA Tuxedo online documentation.

When the client or server application issues a request on an object via an object reference, the BEA Tuxedo server application instantiates the object specified by the object reference, if the object is not already active in memory. (Note that a request always maps to a specific operation invocation on an object.)

Instantiating an object typically involves the server application initializing the object's state, which may include having the object's state read from durable storage, such as a database.

The object contains all the data necessary to do the following:

- Execute the object's operations.
- Store the object's state in durable storage when the object is no longer needed.

How a CORBA Object Comes into Being

The data that makes up a CORBA object may have its origin as a record in a database. The record in the database is the persistent, or durable, state of the object. This record becomes accessible via a CORBA object in an BEA Tuxedo domain when the following sequence has occurred:

1. The server application's factory creates a reference for the object. The object reference includes information about how to locate the record in the database.
2. Using the object reference created by the factory, the client application issues a request on the object.
3. The object is instantiated. The object is instantiated by the TP Framework by invoking the `Server::create_servant` method, which exists in the Server object.
4. The BEA Tuxedo domain invokes the `activate_object` operation on the object, which causes the record containing state to be read into memory.

Whereas a language object exists only within the boundaries of the execution of the application, a CORBA object may exist across processes and machine systems. The BEA Tuxedo system

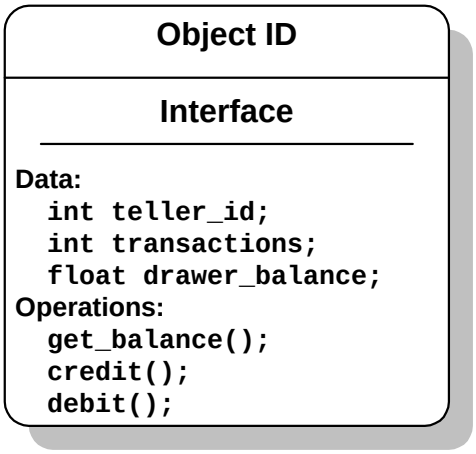
provides the mechanism for constructing an object and for making that object accessible to the application.

The BEA Tuxedo CORBA server application programmer is responsible for writing the code that initializes an object's state and the code that handles that object's state after the object is no longer active in the application. If the object has data in durable storage, this code includes the operations that read from and write to durable storage. For more information about developing server applications, see *Creating CORBA Server Applications* in the BEA Tuxedo online documentation.

Components of a CORBA Object

CORBA objects typically have the following components, shown in the figure that follows:

- An ID, also known as an object ID, or OID
- An interface, which specifies the CORBA object's data and operations



The sections that follow describe each of these object components in detail.

The Object ID

The object ID (OID) associates an object with its state, such as a database record, and identifies the instance of the object. When the factory creates an object reference, the factory assigns an

OID that may be based on parameters that are passed to the factory in the request for the object reference.

Note: The server application programmer must create the factories used in the BEA Tuxedo client/server application. The programmer is responsible for writing the code that assigns OIDs. Factories, and examples of creating them, are discussed in *Creating CORBA Server Applications*.

The BEA Tuxedo system can determine how to instantiate the object by using the following information:

- The OID
- Addressing data in the object reference
- The group ID in the object reference

The Object Interface

The object's interface, described in the application's OMG IDL file, identifies the set of data and operations that can be performed on an object. For example, the interface for a university teller object would identify:

- The data types associated with the object, such as a teller ID, cash in the teller's drawer; and the data managed by the object, such as an account.
- The operations that can be performed on that object, such as obtaining an account's current balance, debiting an account, or crediting an account.

One distinguishing characteristic of a CORBA object is the run-time separation of the interface definition from its data and operations. In a CORBA system, a CORBA object's interface definition may exist in a component called the Interface Repository. The data and operations are specified by the interface definition, but the data and operations exist in the server application process when the object is activated.

The Object's Data

The object's data includes all of the information that is specific to an object class or an object instance. For example, within the context of a university application, a typical object might be a teller. The data of the teller could be:

- An ID
- The amount of cash in the teller's drawer

- The number of transactions the teller has processed during a given interval, such as a day or month

You can encapsulate the object's data in an efficient way, such as by combining the object's data in a structure to which you can get access by means of an attribute. Attributes are a conventional way to differentiate the object's data from its operations.

The Object's Operations

The object's operations are the set of routines that can perform work using the object's data. For example, some of the operations that perform functions using teller object might include:

- `get_balance()`
- `credit()`
- `debit()`

In a CORBA system, the body of code you write for an object's operations is sometimes called the object implementation, which is explained in the next section.

Where an Object Gets Its Operations

As explained in the preceding section, the data that makes up a CORBA object may exist in a record in a database. Alternatively, the data could be established for a CORBA object only when the object is active in memory. This section explains how to write operations for a CORBA object and how to make the operations a part of the object.

The operations you write for a given CORBA object are also known as the object's *implementation*. You can think of the implementation as the code that provides the behavior of the object. When you create an BEA Tuxedo CORBA client/server application, one of the steps you take is to compile the application's OMG IDL file. The OMG IDL file contains statements that describe the application's interfaces and the operations that can be performed on those interfaces.

If you are implementing your server application in C++, one of the several files optionally produced by the IDL compiler is a template for the ***implementation file***. The template for the implementation file contains default constructors and method signatures for your application's objects. The implementation file is where you write the code that implements an object; that is, this file contains the business logic of the operations for a given interface.

The BEA Tuxedo system implements an interface as a CORBA object. The IDL compiler also produces other files, which get built into the BEA Tuxedo CORBA client and server application,

that make sure that the implementation you write for a given object gets properly connected to the correct object data during run time.

This is where the notion of a servant comes in. A servant is an instance of the object class; that is, a servant is an instance of the method code you wrote for each operation in the implementation file. When the BEA Tuxedo CORBA client and server applications are running, and a client request arrives in the server application for an object that is not active -- that is, the object is not in memory -- the following events occur:

1. If no servant is currently available for the needed object, the BEA Tuxedo system invokes the `Server::create_servant` method on the Server object.

   The `Server::create_servant` method is entirely user-written. The code that you write for the `Server::create_servant` method instantiates the servant needed for the request. Your code can use the interface name, which is passed as a parameter to the `Server::create_servant` method, to determine the type of servant that the BEA Tuxedo domain creates.

   The servant that the BEA Tuxedo domain creates is a specific *servant object instance* (it is not a CORBA object), and this servant contains an executable version of the operations you wrote earlier that implement the CORBA object needed for the request.

2. The BEA Tuxedo domain passes control to the servant, and optionally invokes the servant's `activate_object` method, if you have implemented it. Invoking the `activate_object` method gives life to the CORBA object, as follows:

   a. You write the code for the `activate_object` method. The parameter to the `activate_object` method is the string value of the object ID for the object to be activated. You may use the object ID as a key to how to initialize the object.

   b. You initialize the CORBA object's data, which may involve reading state data from durable storage, such as from a record in a database.

   c. The servant's operations become bound to the data, and the combination of those operations and the data establish the activated CORBA object.

   After steps a, b, and c are completed, the CORBA object is said to be activated.

   Implementing the `activate_object` method on an object is optional. For more information about when you want to implement this operation on an object, see *Creating CORBA Server Applications* in the BEA Tuxedo online documentation.

Note: A servant is not a CORBA object. In fact, the servant is represented as a language object. The server performs operations on an object via its servant.

For more information about creating object implementations, see *Creating CORBA Server Applications* in the BEA Tuxedo online documentation.

How Object Invocation Works

Since CORBA objects are meant to function in a distributed environment, OMG has defined an architecture for how object invocation works. A CORBA object can be invoked in one of two ways:

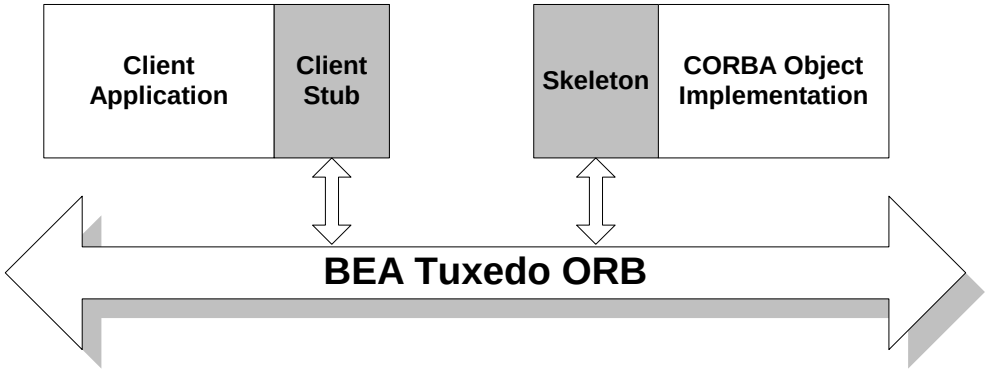
- By means of generated client stubs and skeletons -- sometimes referred to as stub-style invocation.
- By means of the dynamic invocation interface -- referred to as dynamic invocation.

Creating CORBA Client Applications describes how dynamic invocation works. This section describes stub-style invocation, which is simpler to use than dynamic invocation.

When you compile your application's OMG IDL file, one file that the compiler generates is a source file called the ***client stub***. The client stub maps OMG IDL operation definitions for an object type to the operations in the CORBA server application that the BEA Tuxedo system invokes to satisfy a request. The client stub contains code generated during the client application build process that is used in sending the request to the server application. Programmers should never modify the client stub code.

Another file produced by the IDL compiler is the ***skeleton***, which is also a source file. The skeleton contains code used for operation invocations on each interface specified in the OMG IDL file. The skeleton is a map that points to the appropriate code in the CORBA object implementation that can satisfy the client request. The skeleton is connected to both the object implementation and the BEA Tuxedo Object Request Broker.

The following figure shows the client application, the client stub, the skeleton, and the CORBA object implementation:



When a client application sends a request, the request is implemented as an operation on the client stub. When the client stub receives the request, the client stub sends the request to the Object Request Broker (ORB), which then sends the request through the BEA Tuxedo system to the skeleton. The ORB interoperates with the TP Framework and the Portable Object Adapter (POA) to locate the correct skeleton and object implementation.

For more information about generating client stubs and skeletons, see *Creating CORBA Client Applications* and *BEA Tuxedo ATMI C Function Reference* in the BEA Tuxedo online documentation.

CHAPTER 3

Process-Entity Design Pattern

This topic includes the following sections:

About the Process-Entity Design Pattern

The Process-Entity design pattern encapsulates a design solution that incorporates a single process object on the server machine that handles all client application interactions with database records, known as entities. This design pattern is appropriate in situations where a client CORBA or EJB application normally performs multiple interactions with a remote database.

By designing a single CORBA object or EJB on the server machine that represents all the fine-grained data in the database, you can build a BEA Tuxedo CORBA client/server application that provides the following performance benefits:

- Instead of having multiple client interactions with a database, you can have a single process object on the server machine that handles all client requests for database interactions, thus simplifying network traffic.
- The process object can selectively pass data fields to the client, transferring only the necessary data rather than full database records, thus reducing the amount of data sent over the network and improving performance.
- The process object encapsulates access to the database. Clients make invocations on the object, and the object in turn accesses the database.

Increasing Scalability and Resource Utilization

This topic includes the following sections:

Limitations of the Two-tier System

In a conventional two-tier system that presents the database layer as a set of shared data, a pure object-oriented approach would be to represent the database records as shared CORBA objects (in CORBA applications) or entity beans (in EJB applications). However, this approach has the following limitations:

- It does not scale well. As the number of clients increases dramatically, the server machine might be required to manage thousands (or even millions) of database objects, each requiring its own transaction context.

- It does not use network resources efficiently. When database objects are instantiated in the server machine's memory, the entire database object is read into or written from memory regardless of how much data the client application really needs from the object.
- For EJB applications, the conventional way to access a database is via entity beans, where an entity bean represents a row in a database table. However, to access an entity bean, a client application must make two calls: the first call obtains the object reference to the entity bean, and the second call invokes a method on that entity bean. Obtaining the object reference for the client is an expensive operation, particularly in high-volume enterprise applications.

Advantages of the Process-Entity Design Pattern

However, if you design a class for the process object on the server machine that does database interactions on behalf of clients, you can overcome these limitations by:

- Reducing the number of CORBA objects or EJBs that need to be managed on the server machine.
- Reducing message traffic.
- For EJB applications, eliminating the need for client applications to obtain an object reference to the entity bean, and avoiding the use of fine-grain (single-row) entity beans.

Applicability

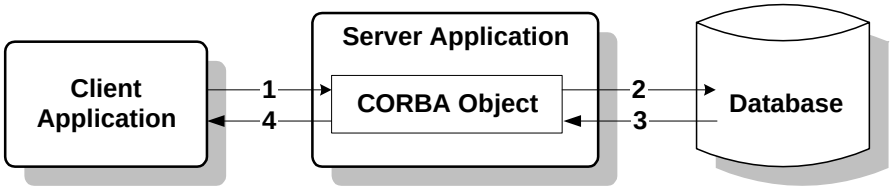
This topic includes the following sections:

The Process-Entity design pattern is almost universally applicable in enterprise-class, mission-critical applications. It is appropriate for situations in which a client application needs to interact with database records stored on a server machine.

Request Flow in CORBA Applications

Figure 3-1 shows the basic design of the Process-Entity design pattern in a CORBA application.

Figure 3-1 CORBA Process-Entity Design Pattern



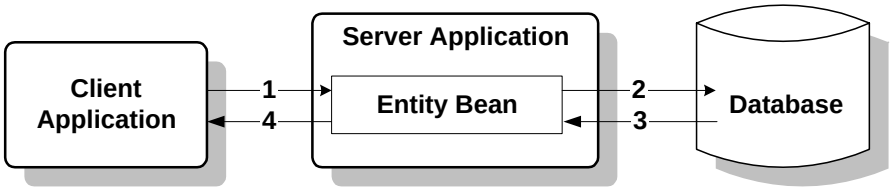
This process flows in the following sequence:

1. The client application issues a request to the CORBA process object to access database entities.
2. The CORBA object submits a request to the database.
3. The database returns a response to the CORBA object.
4. The CORBA object returns a response to the client that contains only the subset of database information that the client requires.

Request Flow in EJB Applications

Figure 3-2 shows the basic design of the Process-Entity design pattern in an EJB application.

Figure 3-2 EJB Process-Entity Design Pattern



This process flows in the following sequence:

1. The client application issues a request to the entity bean, using RMI on IIOP, to access database entities.
2. The entity bean submits a request to the database.
3. The database returns a response to the entity bean.

4. The entity bean returns a response to the client that contains only the subset of database information that the client requires.

Participants

The client application obtains a reference to the process object from a factory (for CORBA applications) or the home interface (for EJB applications). The process object implements all the interactions with the database. Database records (entities) are retrieved when needed to handle client invocations on the process object. Operations on the process object return specific data fields to the client application, which then performs all the required processing on that data.

Other Considerations

You should design the process object to pass the minimum amount of information actually needed by a particular client request. Implement the operations on a process object so that the operations do as much “dense” processing as possible. Design your clients applications so that they do not invoke more than one process operation to get the data they need to accomplish a task.

If more than one operation needs to be invoked, design the process object so that the additional invocations are done by the process object on the database, and not by the client application on the process object. This reduces the number of invocations that the client application sends over the network. When the client application needs to make serial invocations on a process object, make the process object stateful. For more information about making objects stateful, see *Creating CORBA Server Applications* in the BEA Tuxedo online documentation.

For CORBA applications, avoid the use of attributes in your OMG IDL. Attributes are expensive to retrieve over the network. Instead, implement an operation on the process object that returns a data structure containing all the values your client application is likely to need for an operation.

Related Concepts

- SmallTalk MVC (Model-View-Controller) design pattern.
- Flyweight design pattern, *Object-Oriented Design Patterns*, by Gamma et al.

CHAPTER 4

Client Data Caching Design Pattern

This chapter describes the CORBA Client Data Caching design pattern. The purpose of this design pattern is to make persistent state information from the server available locally to the client for data-intensive processing. This way, the CORBA client application does not need to make repeated invocations to the server application to retrieve data.

Motivation

This design pattern addresses the scalability and performance of distributed client/server applications. The overhead associated with remote invocations to retrieve attributes of a CORBA object may be quite high, depending on system load and other factors. Also, exposing persistent data records as CORBA objects tends to create applications that do not scale well because of the potentially large number of simultaneously active objects that must be managed by the system. Client application processing that is either data-intensive or that requires user input (for example, editing fields) can slow down both the client application and the system if multiple remote invocations must be made to retrieve data.

Applicability

This design pattern is appropriate in situations where the CORBA process object needs to pass a large amount of data to the client application for its use. The local language object on the client application becomes a container for the data, and its constructor is used to populate the local object state.

You implement this design pattern in the client application, which creates a local language object, referred to in this chapter as the `DataObject`. The server application implements a CORBA

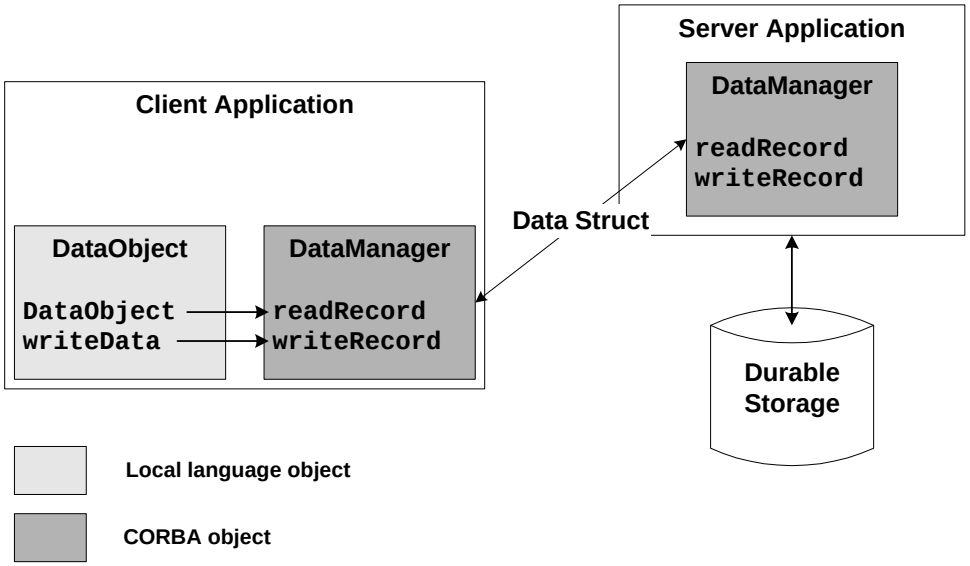
process object that interacts with entities in persistent storage. This CORBA object is referred to in this chapter as the `DataManager` object.

The OMG IDL for the `DataManager` CORBA object defines a data structure that is used for transferring data between the client and server applications. (This design pattern assumes "optimistic locking," meaning that the data managed by the server application is not locked for update, and that it is hoped that no other server processes modify the data while the client application uses its local copy.)

When the client application instantiates the local `DataObject`, that object's constructor invokes an operation on the `DataManager` CORBA object, which passes a data structure back to the `DataObject`. The `DataObject` populates its class variables with the passed data.

If the client application needs to pass modified state back to the server machine, the client application invokes the `DataObject::writeData()` method, which, in turn, invokes the `writeRecord()` operation on the `DataManager` CORBA object. In this invocation, the data structure is passed as a parameter to the `writeRecord()` operation. The `DataManager` CORBA object makes the appropriate updates to durable storage.

The following figure illustrates how the CORBA Client Data Caching design pattern works.



In the preceding figure:

1. The `DataObject` constructor invokes the `readRecord()` operation on the `DataManager` CORBA object, and uses the returned data structure to initialize its local state.
2. The client application may modify the local state of the `DataObject` instance.
3. To pass modified state back to the `DataManager` CORBA object, the client application invokes the `DataObject::writeData()` operation, passing a data structure containing the modified data.

Participants

The `DataObject` methods read and write data by invoking operations on the `DataManager` CORBA object.

Other Considerations

The data structure passed to the client application should be designed to provide the minimal set of data required for an operation. If a large amount of data is involved, it may be more efficient to provide multiple data structures with a subset of fields required for each operation. Operations on the CORBA process object should be designed to involve only the subset of data needed for each operation; this helps reduce network traffic.