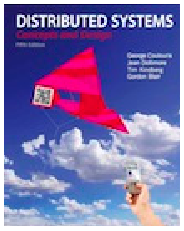

Distributed Systems: Concepts and Design

Chapter 8 Exercise Solutions

8.1 The Task Bag is an object that stores (*key*, *value*) pairs. A *key* is a string and a *value* is a sequence of bytes. Its interface provides the following remote methods:

pairOut, with two parameters through which the client specifies a *key* and a *value* to be stored.

pairIn, whose first parameter allows the client to specify the *key* of a pair to be removed from the Task Bag. The *value* in the pair is supplied to the client via a second parameter. If no matching pair is available, an exception is thrown.

readPair, which is the same as *pairIn* except that the pair remains in the Task Bag.

Use CORBA IDL to define the interface of the Task Bag. Define an exception that can be thrown whenever any one of the operations cannot be carried out. Your exception should return an integer indicating the problem number and a string describing the problem. The Task Bag interface should define a single attribute giving the number of tasks in the bag.

8.1 Ans.

Note that sequences must be defined as *typedefs*. *Key* is also a *typedef* for convenience.

```
typedef string Key;
typedef sequence<octet> Value;
interface TaskBag {
    readonly attribute long numberOfTasks;
    exception TaskBagException { long no; string reason; };
    void pairOut (in Key key, in Value value) raises (TaskBagException);
    void pairIn (in Key key, out Value value) raises (TaskBagException);
    void readPair (in Key key, out Value value) raises (TaskBagException);
};
```

8.2 Define an alternative signature for the methods *pairIn* and *readPair*, whose return value indicates when no matching pair is available. The return value should be defined as an enumerated type whose values can be *ok* and *wait*. Discuss the relative merits of the two alternative approaches. Which approach would you use to indicate an error such as a key that contains illegal characters?

8.2 Ans.

```
enum status { ok, wait};
status pairIn (in Key key, out Value value);
status readPair (in Key key, out Value value);
```

It is generally more complex for a programmer to deal with an exception that an ordinary return because exceptions break the normal flow of control. In this example, it is quite normal for the client calling *pairIn* to find that the server hasn't yet got a matching pair. Therefore an exception is not a good solution.

For the key containing illegal characters it is better to use an exception: it is an error (and presumably an unusual occurrence) and in addition, the exception can supply additional information about the error. The client must be supplied with sufficient information to recognise the problem.

8.3 Which of the methods in the Task Bag interface could have been defined as a *oneway* operation? Give a general rule regarding the parameters and exceptions of *oneway* methods. In what way does the meaning of the *oneway* keyword differ from the remainder of IDL?

8.3 Ans.

A *oneway* operation cannot have *out* or *inout* parameters, nor must it raise an exception because there is no reply message. No information can be sent back to the client. This rules out all of the Task Bag operations. The *pairOut* method might be made one way if its exception was not needed, for example if the server could guarantee to store every pair sent to it. However, *oneway* operations are generally implemented with *maybe* semantics, which is unacceptable in this case. Therefore the answer is *none*.

General rule. Return value *void*. No *out* or *inout* parameters, no user-defined exceptions.

The rest of IDL is for defining the interface of a remote object. But *oneway* is used to specify the required quality of delivery.

8.4 The IDL *union* type can be used for a parameter that will need to pass one of a small number of types. Use it to define the type of a parameter that is sometimes empty and sometimes has the type *Value*.

8.4 Ans.

```
union ValueOption switch (boolean){
    case TRUE: Value value;
};
```

When the value of the tag is TRUE, a *Value* is passed. When it is FALSE, only the tag is passed.

8.5 In Figure 8.2 the type *All* was defined as a sequence of fixed length. Redefine this as an array of the same length. Give some recommendations as to the choice between arrays and sequences in an IDL interface.

8.5 Ans.

```
typedef Shape All[100];
```

Recommendations

- if a fixed length structure then use an array.
- if variable length structure then use a sequence
- if you need to embed data within data, then use a sequence because one may be embedded in another
- if your data is sparse over its index, it may be better to use a sequence of <index, value> pairs.

8.6 The Task Bag is intended to be used by cooperating clients, some of which add pairs (describing tasks) and others of which remove them (and carry out the tasks described). When a client is informed that no matching pair is available, it cannot continue with its work until a pair becomes available. Define an appropriate callback interface for use in this situation.

8.6 Ans.

This callback can send the value required by a *readPair* or *pairIn* operation. Its method should not be a *oneway* as the client depends on receiving it to continue its work.

```
interface TaskBagCallback{
    void data(in Value value);
}
```

8.7 Describe the necessary modifications to the Task Bag interface to allow callbacks to be used.

8.7 Ans.

The server must allow each client to register its interest in receiving callbacks and possibly also deregister. These operations may be added to the TaskBag interface or to a separate interface implemented by the server.

int register (in TaskBagCallback callback);
void deregister (in int callbackId);

See the discussion on callbacks in Chapter 5. (page 223)

8.8 Which of the parameters of the methods in the Task Bag interface are passed by value and which are passed by reference?

8.8 Ans.

In the original interface, all of the parameters are passed by value.
The parameter of *register* is passed by reference. (and that of *deregister* by value)

8.9 Use the Java IDL compiler to process the interface you defined in Exercise 8.1. Inspect the definition of the signatures for the methods *pairIn* and *readPair* in the generated Java equivalent of the IDL interface. Look also at the generated definition of the holder method for the *value* argument for the methods *pairIn* and *readPair*. Now give an example showing how the client will invoke the *pairIn* method, explaining how it will acquire the value returned via the second argument.

8.9 Ans.

The Java interface is:

```
public interface TaskBag extends org.omg.CORBA.Object {
    void pairOut(in Key p, in Value value);
    void pairIn(in Key p, out ValueHolder value);
    void readPair(in Key p, out ValueHolder value);
    int numberOfTasks();
}
```

The class *ValueHolder* has an instance variable

```
public Value value;
```

and a constructor

```
public ValueHolder(Value __arg) {
    value = __arg;
}
```

The client must first get a remote object reference to the TaskBag (see Figure 20.5). probably via the naming service.

```
 ...
TaskBag taskBagRef = TaskBagHelper.narrow(taskBagRef.resolve(path));
Value aValue;
taskbagRef.pairIn(“Some key”, new ValueHolder(aValue));
```

The required value will be in the variable *aValue*.

8.10 Give an example to show how a Java client will access the attribute giving the number of tasks in the Task Bag object. In what respects does an attribute differ from an instance variable of an object?

8.10 Ans.

Assume the IDL attribute was called *numberOfTasks* as in the answer to Exercise 20.1. Then the client uses the method *numberOfTasks* to access this attribute. e.g.

```
taskbagRef.numberOfTasks();
```

Attributes indicate methods that a client can invoke in a CORBA object. They do not allow the client to make any assumption about the storage used in the CORBA object, whereas an instance variable declares the type of a variable. An attribute may be implemented as a variable or it may be a method that calculates the result. Either way, the client invokes a method and the server implements it.

8.11 Explain why the interfaces to remote objects in general and CORBA objects in particular do not provide constructors. Explain how CORBA objects can be created in the absence of constructors.

8.11 Ans.

If clients were allowed to request a server to create instances of a given interface, the server would need to provide its implementation. It is more effective for a server to provide an implementation and then offer its interface.

CORBA objects can be created within the server:

1. the server (e.g. in the *main* method) creates an instance of the implementation class and then exports a remote object reference for accessing its methods.
2. A factory method (in another CORBA object) creates an instance of the implementation class and then exports a remote object reference for accessing its methods.

8.12 Redefine the Task Bag interface from Exercise 8.1 in IDL so that it makes use of a *struct* to represent a *Pair*, which consists of a *Key* and a *Value*. Note that there is no need to use a *typedef* to define a *struct*.

8.12 Ans.

```
typedef string Key;
typedef sequence<octet> Value;
struct Pair {
    Key key;
    Value value;
  };
interface TaskBag {
readonly attribute int numberOfTasks;
exception TaskBagException { int no; string reason; };
void pairOut (in Pair) raises (TaskBagException);
    // pairIn and readPair might use the pair, or could be left unchanged
};
```

8.13 Discuss the functions of the implementation repository from the point of view of scalability and fault tolerance.

8.13 Ans.

The implementation repository is used by clients to locate objects and activate implementations. A remote object reference contains the address of the IR that manages its implementation.

An IR is shared by clients and servers within some location domain.

To improve scalability, several IRs can be deployed, with the implementations partitioned between them (the object references locate the appropriate IR). Clients can avoid unnecessary requests to an IR if they parse the remote object reference to discover whether they are addressing a request to an object implementation already located.

From the fault tolerance point of view, an IR is a single point of failure. Information in IRs can be replicated - note that a remote object reference can contain the addresses of several IRs. Clients can try them in turn if one is unobtainable. The same scheme can be used to improve availability.

8.14 To what extent may CORBA objects be migrated from one server to another?

8.14 Ans.

CORBA persistent IORs contain the address of the IR used by a group of servers. That IR can locate and activate CORBA objects within any one of those servers. Therefore, it will still be able deal with CORBA objects that migrate from one server in the group to another. But the object adapter name is the key for the implementation in the IR. Therefore all of the objects in one server must move together to another server. This could be modified by allowing groups of objects within each server to have separate object adapters and to be listed under different object adapter names in the IR.

8.15 Explain carefully how component-based middleware in general and EJB in particular can overcome the key limitations of distributed object middleware. Provide examples to illustrate your answer.

8.15 Ans.

The problems and associated solutions are described below.

Implicit dependencies: Component-based middleware get round this issue by having a complete contract with the environment (and other components) through both required and provided interfaces. Required interfaces in particular act as declarations of dependencies on other components that need to be there and bound to this component for it to function correctly. This eliminates implicit dependencies. In EJB, this is achieved through the mechanism of dependency injection. As an example, consider making calls to a transaction manager. In distributed object middleware such calls may well be embedded in the code of an object and this is not visible from outside the encapsulation. In EJB, this would be declared as a dependency, as in the call the following call taken from Section 8.5.1:

@Resource javax.transaction.UserTransaction ut;

Programming complexity: ?Programming complexity is concerned with the need to make low level calls to the underlying middleware. This is particularly true in CORBA where the Portable Object Adaptor (POA) and ORB Core, for example, offers a relatively sophisticated interfaces for areas such as the creation and management of object references, the management of object lifecycles, activation and passivation policies, the management of persistent state and policies for mappings to underlying platform resources such as threads. Components hide these interfaces by doing such management within the container abstraction. EJB does not have to deal with the complexities of CORBA but still significantly simplifies lifecycle management, for example.

Lack of separation of distribution concerns: There is additional complexity in managing calls to distributed systems services such as security or transaction services. Again, this is hidden from the programmer through the container abstraction. In EJB, this is particularly easy to deal with through a combination of configuration by exception and the use of annotations to declare requirements more declaratively. An example is providing support for transaction management which, in the simplest case (container-managed transactions), can be set up with simple annotations such as:

@TransactionManagement (CONTAINER)

No support for deployment: All component technologies provide support for deployment through the packaging of components with associated architectural descriptions and deployment descriptors. Tools are provided to interpret such packaging and deploy the associated configurations. In EJB, jar files fulfil this purpose.

8.16 Discuss whether the EJB architecture would be suitable to implement a massively multiplayer online game (an application domain initially introduced in Section 1.2.2). What would be the strengths and weaknesses of using EJB in this domain?

8.16 Ans.

The EJB architecture makes it easier to develop distributed applications by hiding much of the complexity from the programmer. It does this by imposing a particular architectural pattern on applications, that is one where a potentially large number of clients access fairly coarse granularity resources and this needs protection in terms of security and transaction management, a pattern which works best for applications such as eCommerce or banking applications that fit naturally this style of architecture.

Massively multiplayer online games are quite a different class of application. For example, they require low latency interaction with the world views and also may tolerate weak consistency to support such low latency interaction. It is likely that the container-style interaction would not be right for this application domain although it would provide some support for the development of such applications. More importantly, it is likely that the non-functional properties (and the associated policies) provided for containers would not be right for this domain, for example for transaction management (multiplayer online games would not require full transaction management). There would also be question marks over whether an EJB implementation would scale in the right way to large number of online players, all requiring low latency interactions.

8.17 Would Fractal be a more suitable implementation choice for the MMOG domain? Justify your answer.

8.17 Ans.

It is much more likely that a successful implementation could be achieved using Fractal than EJB, the reasons being that: (1) Fractal is more lightweight and hence would lend itself to achieving good performance and also, (2) while Fractal can support a container-like abstraction, it is again more lightweight and much less prescriptive in what this offers, and would allow more tailored policies to be developed, for example supporting weaker forms of consistency.

8.18 Explain how the container-based philosophy could be adopted to provide migration transparency for distributed components.

8.18 Ans.

As a reminder, migration (or mobility) transparency is concerned with hiding the movement of distributed system entities from users or programmers. Containers operate by intercepting incoming invocations and making a series of calls to implement the desired level of transparency before optionally passing on the invocation to the component. To implement migration transparency, a container can use a location service to map objects (or in this case components) on to physical locations. (and keep track of them if they move). For example, Section 5.4.2 explains how a location service is implemented in Clouds and Emerald.

8.19 How would you achieve the same effect in Fractal?

8.19 Ans.

Exactly the same effect can be achieved by using the interception mechanism as provided in controllers. Fractal, with its open and configurable approach, also lends itself to the introduction or new levels of transparency.

8.20 Consider the implementation of Java RMI as a composite binding in Fractal. Discuss the extent to which such a binding can be both configurable and reconfigurable.

8.20 Ans.

Composite binding are like any other configuration in Fractal and can themselves be configured and reconfigured. That is, a composite binding has an associated software architecture in terms of components and bindings. It is therefore possible to provide a component-based implementation of Java RMI which teases apart the different aspects of the implementation, for example components responsible for providing proxies, naming, communication, location, and so on. The granularity of the composition can also be varied, for example, the communication module could further be broken down into different aspects such as the underlying transport protocol and the module implementing the required invocation semantics. This approach clearly lends itself to a more configurable approach. For example, rather than offering a range of semantics in an implementation (at-least-once, at-most-once and exactly-once) a given deployment could be configured to offer one, as required by that context. Similarly, the transport protocol could be configured to TCP or to an alternative transport mechanism, for example optimized for an ad hoc networking environment. Similarly, the approach lends itself to reconfiguration, for example changing the invocation semantics or transport protocol at runtime.