



# Distributed Systems: Concepts and Design

**Edition 3**

**By George Coulouris, Jean Dollimore and Tim Kindberg**  
**Addison-Wesley, ©Pearson Education 2001.**

## Chapter 17 Exercise Solutions

- 17.1 The Task Bag is an object that stores pairs of (key and value). 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*: 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.

*17.1 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);
};
```

- 17.2 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?

*17.2 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 than 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.

- 
- 17.3 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?

17.3 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.

- 
- 17.4 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*.

17.4 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.

- 
- 17.5 In Figure 17.1 the type *All* was defined as a sequence of a 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.

17.5 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.

- 
- 17.6 The Task Bag is intended to be used by cooperating clients, some of which add pairs (describing tasks) and others 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.

17.6 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);
}
```

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

17.7 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 200)

- 
- 17.8 Which of the parameters of the methods in the TaskBag interface are passed by value and which are passed by reference?

*17.8 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)

---

- 17.9 Use the Java IDL compiler to process the interface you defined in Exercise 17.1. 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.

*17.9 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 17.5). 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*.

---

- 17.10 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?

*17.10 Ans.*

Assume the IDL attribute was called *numberOfTasks* as in the answer to Exercise 17.1. 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.

---

17.11 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.

*17.11 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.

---

17.12 Redefine the Task Bag interface from Exercise 17.1 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*.

*17.12 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
};
```

---

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

*17.13 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.

---

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

*17.14 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.

Also, CORBA objects cannot move to a server that uses a different IR. It would be possible for servers to move and to register with a new IR, but then there are issues related to finding it from the old location domain, which would need to have forwarding information.

- 
- 17.15 Discuss the benefits and drawbacks of the two-part names or *NameComponents* in the CORBA naming service.

*17.15 Ans.*

A *NameComponent* contains a *name* and a *kind*. The *name* field is the name by which the component is labelled (like a name component in a file or DNS name).

The *kind* field is intended for describing the name. It is not clear that this has any useful function. The use of two parts for each name complicates the programming interface. Without it, each name component could be a simple string.

- 
- 17.16 Give an algorithm that describes how a multipart name is resolved in the CORBA naming service. A client program needs to resolve a multipart name with components “A”, “B” and “C”, relative to an initial naming context. How would it specify the arguments for the *resolve* operation in the naming service?

*17.16 Ans.*

Given a multipart name, *mn*[] with components *mn*[0], *mn*[1], *mn*[2], ...*mn*[N], starting in context C. Note *mn* is of type *Name* (a sequence of *NameComponents*).

The method *resolve* returns the *RemoteObjectRef* of the object (or context) in the context matched by the name, *mn*[],. There are several ways in which *resolve* may fail:

- i) If the name *mn* is longer than the path in the naming graph, then when *mn*[*i*] (*i*<*n*) is looked up, we get an object and throw a *NameNotFound* exception with reason *NonContext*. Assume that *length(mn)* > 0 to start with.
- ii) If the part *nm*[*i*] does not match a name in the current context, throw a *NameNotFound* exception with reason *missingName*.
- iii) if a *RemoteObjectRef* turns out not to refer to an object or context at all throw a *NameNotFound* exception with reason *invalidRef*.

A more sophisticated answer might return the remainder of the path with the exception. The algorithm can be defined as follows:

```
RemoteObjectRef resolve(C, Name mn[]) {
    RemoteObjectRef ref = lookInContext(C, first(mn));
    if(ref==null) throw NameNotFoundException (missingName);
    else if(length(mn) == 1) return ref;
    else if(type(ref) == object) throw NameNotFoundException(NonContext)
    else resolve( C', tail(mn));
}
```

where *lookInContext(C, name)* looks up the name component, *name* in context *C* and returns a *RemoteObjectRef* of an object or a context (or null if the name is not found).

Name name;

```
NamingContext C = resolve_initial_references("NameService");
name[0] = new NameComponent("A","");
name[1] = new NameComponent("B","");
name[2] = new NameComponent("C","");
C.resolve(name);
```

- 
- 17.17 A virtual enterprise consists of a collection of companies who are cooperating with one another to carry out a particular project. Each company wishes to provide the others with access to only those of its CORBA objects relevant to the project. Describe an appropriate way for the group to federate their CORBA Naming Services.

*17.17 Ans.*

The solution should suggest that each company manages its own naming graph and decides which portion of it should be made available for sharing by the other companies in the virtual enterprise. Each company provides the others with a remote object reference to the shared portion of their naming graph (a remote object reference may be converted to a string and passed to the others e.g. by email). Each the companies provides names that link the remote object reference to the set of CORBA objects held by the others via its naming graph. The companies might agree on a common naming scheme.

- 
- 17.18 Discuss how to use directly connected suppliers and consumers of the CORBA event service in the context of the shared whiteboard application. The *PushConsumer* and *PushSupplier* interfaces are defined in IDL as follows:

```
interface PushConsumer {
    void push(in any data) raises (Disconnected);
    void disconnect_push_consumer();
}

interface PushSupplier {
    void disconnect_push_supplier();
}
```

Either the supplier or the consumer may decide to terminate the event communication by calling *disconnect\_push\_supplier()* or *disconnect\_push\_consumer()* respectively.

*17.18 Ans.*

The shared whiteboard application is described on pages 195-6.

Choose whether to use *push* or *pull* model. We use the *push* model in which the supplier (object of interest) initiates the transfer of notifications to subscribers (consumers). Since we have only one type of event in this application (a new *GraphicalObject* has been added at the server), we use generic events.

In the push model, the consumer (the whiteboard client) must implement the *PushConsumer* interface. It could do this by providing an implementation of a servant with the following interface (including the *push* method).

```
interface WhiteboardConsumer: PushConsumer{
    push(in any data) raises (Disconnected);
};
```

The implementation will make *push* do whatever is needed - to get the latest graphical objects from the server. The client creates an instance of *WhiteboardConsumer* and informs the server about it.

The supplier (our server) provides an operation allowing clients to inform it that they are consumers. For example:

```
e.g. addConsumer(WhiteboardConsumer consumer)
    // add this consumer to a vector of WhiteboardConsumers
```

Whenever a new graphical object is created, the server calls the push operation in order to send an event to the client, passing the event data (version number) as argument. For example (*Any* is Java's representation of CORBA any):

```
Any a;
int version;
// assign the int version to the Any a
// repeat for all consumers in the vector
consumer.push(a);
```

- 17.19 Describe how to interpose an Event Channel between the supplier and the consumers in your solution to 17.13. An event channel has the following IDL interface:

```
interface EventChannel {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
};
```

where the interfaces *SupplierAdmin* and *ConsumerAdmin*, which allow the supplier and the consumer to get proxies are defined in IDL as follows:

```
interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
    ---
};

interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier();
    ---
};
```

The interface for the proxy consumer and proxy supplier are defined in IDL as follows:

```
interface ProxyPushConsumer : PushConsumer{
    void connect_push_supplier (in PushSupplier supplier)
        raises (AlreadyConnected);
};

interface ProxyPushSupplier : PushSupplier{
    void connect_push_consumer (in PushConsumer consumer)
        raises (AlreadyConnected);
};
```

What advantage is gained by the use of the event channel?

*17.19 Ans.*

We need to assume something to create an event channel. (E.g. the specification of the Event service has an example in the Appendix showing the creation of an event channel from a factory.

```
EventChannelFactory ecf = ...
ecf.create_eventchannel();
```

The event channel can be created by the whiteboard server and registered with the Naming Service so that clients can get a remote reference to it. More simply, the clients can get a remote reference to the event channel by means of an RMI method in the server interface.

The event channel will carry out the work of sending each event to all of the consumers.

We require one proxy push consumer that receives all notifications from the whiteboard server. The event channel forwards it to all of the proxy push suppliers - one for each client.

The whiteboard server (as an instance of *PushSupplier*) gets a proxy consumer from the event channel and the connects to it by providing its own object reference.

```
SupplierAdmin sadmin = ec.for_suppliers();
ProxyPushConsumer ppc = sadmin.obtain_push_consumer();
```

It connects to the proxy consumer by providing its own object reference

```
ppc.connect_push_supplier(this)
```

The supplier pushes data to the event channel, which pushes it on to the consumers. It may supply events to one or more consumers.

The *newShape* method of *shapeListServant* (Figure 17.3) will use the *push* operation to inform the event channel, each time a new *GraphicalObject* is added. e.g.

```
ppc.push(version);
```

As before, each client implements a CORBA object with a *PushConsumer* interface. It then gets a proxy supplier from the event channel and then connects to it by providing its own object reference.

```
ConsumerAdmin cadmin = ec.for_consumers();  
ProxyPushSupplier pps = cadmin.obtain_push_supplier();
```

It should connect to the proxy supplier by providing its own object reference  
*pps.connect\_push\_supplier(this);*

As before, the client receives notifications via the *push* method in its *PushConsumer* interface.

The advantage of using an event channel is that the whiteboard server does not need to know how many clients are connected nor to ensure that notifications are sent to all of them, using the appropriate reliability characteristics.