4.3.2 Java object serialization

In Java RMI, both objects and primitive data values may be passed as arguments and results of method invocations. An object is an instance of a Java class. For example, the Java class equivalent to the *Person struct* defined in CORBA IDL might be:

```
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }
    //followed by methods for accessing the instance variables
}
```

The above class states that it implements the *Serializable* interface, which has no methods. Stating that a class implements the *Serializable* interface (which is provided in the *java.io* package) has the effect of allowing its instances to be serialized.

In Java, the term *serialization* refers to the activity of flattening an object or a connected set of objects into a serial form that is suitable for storing on disk or transmitting in a message, for example as an argument or result of an RMI. Deserialization consists of restoring the state of an object or a set of objects from their serialized form. It is assumed that the process that does the deserialization has no prior knowledge of the types of the objects in the serialized form. Therefore, some information about the class of each object is included in the serialized form. This information enables the recipient to load the appropriate class when an object is deserialized.

The information about a class consists of the name of the class and a version number. The version number is intended to change when major changes are made to the class. It can be set by the programmer or calculated automatically as a hash of the name of the class, its instance variables, methods and interfaces. The process that deserializes an object can check that it has the correct version of the class.

Java objects can contain references to other objects. When an object is serialized, all the objects that it references are serialized together with it to ensure that when the object is reconstructed, all of its references can be fulfilled at the destination. References are serialized as *handles* – in this case, the handle is a reference to an object within the serialized form, for example the next number in a sequence of positive integers. The serialization procedure must ensure that there is a 1-1 correspondence between object references and handles. It must also ensure that each object is written once only – on the second or subsequent occurrence of an object, the handle is written instead of the object.

To serialize an object, its class information is written out, followed by the types and names of its instance variables. If the instance variables belong to new classes, then their class information must also be written out, followed by the types and names of their instance variables. This recursive procedure continues until the class information and types and names of instance variables of all of the necessary classes have been written

Figure 4.9 Indication of Java serialized form

Serialized values

Person	8-byte version number		h0
3	int year	java.lang.String name:	java.lang.String place:
1934	5 Smith	6 London	h1

Explanation

class name, version number

number, type and name of instance variables

values of instance variables

The true serialized form contains additional type markers; h0 and h1 are handles

out. Each class is given a handle, and no class is written more than once to the stream of bytes – the handles being written instead where necessary.

The contents of the instance variables that are primitive types, such as integers, chars, booleans, bytes and longs, are written in a portable binary format using methods of the *ObjectOutputStream* class. Strings and characters are written by its method called *writeUTF* using Universal Transfer Format (UTF), which enables ASCII characters to be represented unchanged (in one byte), whereas Unicode characters are represented by multiple bytes. Strings are preceded by the number of bytes they occupy in the stream.

As an example, consider the serialization of the following object:

Person p = new Person("Smith", "London", 1934);

The serialized form is illustrated in Figure 4.9, which omits the values of the handles and of the type markers that indicate the objects, classes, strings and other objects in the full serialized form. The first instance variable (1934) is an integer that has a fixed length; the second and third instance variables are strings and are preceded by their lengths.

To make use of Java serialization, for example to serialize the *Person* object, create an instance of the class *ObjectOutputStream* and invoke its *writeObject* method, passing the *Person* object as argument. To deserialize an object from a stream of data, open an *ObjectInputStream* on the stream and use its *readObject* method to reconstruct the original object. The use of this pair of classes is similar to the use of *DataOutputStream* and *DataInputStream* illustrated in Figures 4.5 and 4.6.

Serialization and deserialization of the arguments and results of remote invocations are generally carried out automatically by the middeware, without any participation by the application programmer. If necessary, programmers with special requirements may write their own version of the methods that read and write objects. To find out how to do this and to get further information about serialization in Java, read the tutorial on object serialization [java.sun.com II]. Another way in which a programmer may modify the effects of serialization is by declaring variables that should not be serialized as *transient*. Examples of things that should not be serialized are references to local resources such as files and sockets.

The use of reflection ◊ The Java language supports *reflection* – the ability to enquire about the properties of a class, such as the names and types of its instance variables and methods. It also enables classes to be created from their names, and a constructor with given argument types to be created for a given class. Reflection makes it possible to do

serialization and deserialization in a completely generic manner. This means that there is no need to generate special marshalling functions for each type of object as described above for CORBA. To find out more about reflection, see Flanagan [1997].

Java object serialization uses reflection to find out the class name of the object to be serialized and the names, types and values of its instance variables. That is all that is needed for the serialized form.

For deserialization, the class name in the serialized form is used to create a class. This is then used to create a new constructor with argument types corresponding to those specified in the serialized form. Finally, the new constructor is used to create a new object with instance variables whose values are read from the serialized form.

4.3.3 Remote object references

When a client invokes a method in a remote object, an invocation message is sent to the server process that hosts the remote object. This message needs to specify which particular object is to have its method invoked. A *remote object reference* is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked. Chapter 5 explains that remote object references are also passed as arguments and returned as results of remote method invocations, that each remote object has a single remote object reference and that remote object references can be compared to see whether they refer to the same remote object. We now discuss the external representation of remote object references.

Remote object references must be generated in a manner that ensures uniqueness over space and time. In general, there may be many processes hosting remote objects, so remote object references must be unique among all of the processes in the various computers in a distributed system. Even after the remote object associated with a given remote object reference is deleted, it is important that the remote object reference is not reused, because its potential invokers may retain obsolete remote object references. Any attempt to invoke a deleted object should produce an error rather than allow access to a different object.

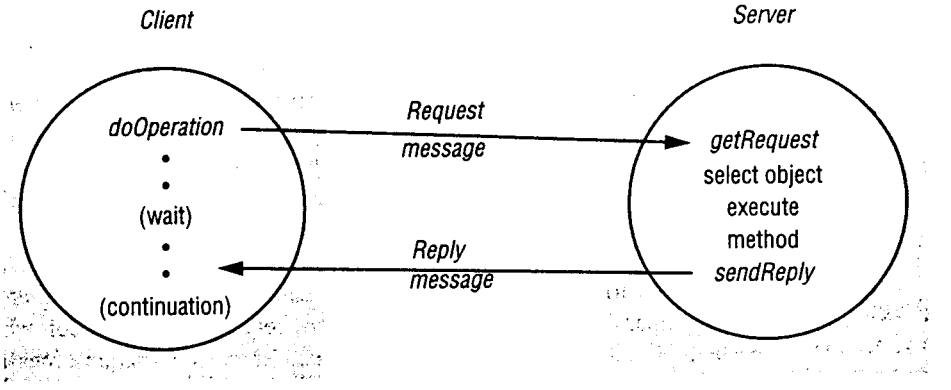
There are several ways to ensure that a remote object reference is unique. One way is to construct a remote object reference by concatenating the Internet address of its computer and the port number of the process that created it with the time of its creation and a local object number. The local object number is incremented each time an object is created in that process.

The port number and time together produce a unique process identifier on that computer. With this approach, remote object references might be represented with a

Figure 4.10 Representation of a remote object reference

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Figure 4.11 Request-reply communication



format such as that shown in Figure 4.10. In the simplest implementations of RMI, remote objects live only in the process that created them and survive only as long as that process continues to run. In such cases, the remote object reference can be used as an address of the remote object. In other words, invocation messages are sent to the Internet address in the remote reference and to the process on that computer using the given port number.

To allow remote objects to be relocated in a different process on a different computer, the remote object reference should not be used as the address of the remote object. Section 17.2.4 discusses a form of remote object reference that allows objects to be activated in different servers throughout its lifetime.

The last field of the remote object reference shown in Figure 4.10 contains some information about the interface of the remote object, for example the interface name. This information is relevant to any process that receives a remote object reference as an argument or result of a remote invocation, because it needs to know about the methods offered by the remote object. This point is explained again in Section 5.2.5.

4.4 Client-server communication

This form of communication is designed to support the roles and message exchanges in typical client-server interactions. In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server. It can also be reliable because the reply from the server is effectively an acknowledgement to the client. Asynchronous request-reply communication is an alternative that may be useful in situations where clients can afford to retrieve replies later – see Section 6.5.2.

The client-server exchanges are described in the following paragraphs in terms of the *send* and *receive* operations in the Java API for UDP datagrams, although many current implementations use TCP streams. A protocol built over datagrams avoids unnecessary overheads associated with the TCP stream protocol. In particular:

- acknowledgements are redundant, since requests are followed by replies;