95-702 Distributed Systems

Chapter 4: Inter-process Communications
Middleware layers

This chapter

Applications, services

RMI and RPC

request-reply protocol
marshalling and external data representation

UDP and TCP

Middleware layers
Marshalling and External Data Representation

Messages consist of sequences of bytes.

Interoperability Problems
- Big-endian, little-endian byte ordering
- Floating point representation
- Character encodings (ASCII, UTF-8, Unicode, EBCDIC)

So, we must either:
- Have both sides agree on an external representation or
- Transmit in the sender’s format along with an indication of the format used. The receiver converts to its form.
External Data Representation and Marshalling

*External data representation* – an agreed standard for the representation of data structures and primitive values

*Marshalling* – the process of taking a collection of data items and assembling them into a form suitable for transmission in a message

*Unmarshalling* – is the process of disassembling them on arrival into an equivalent representation at the destination

The marshalling and unmarshalling are intended to be carried out by the middleware layer.
External Data Representation and Marshalling

Quiz:

Suppose we write a Java TCP client and server. And suppose we pass java objects rather than simple characters, would the server interoperate with a .NET client?
Three Important Approaches

To External Data Representation and Marshalling:

CORBA’s CDR binary data may be used by different programming languages.

Java and .Net Remoting Object Serialization are both platform specific (that is, Java on both sides or .Net on both sides) and binary.

XML is a textual format, verbose when compared to binary but more interoperable.
Interoperability

Consider int j = 3;

What does it look like in memory?
00000000000000000000000000000011

How could we write it to the wire?

Little-Endian approach
Write 00000011
Then 00000000
Then 00000000
Then 00000000

Big-Endian Approach
Write 0000000
Then 0000000
Then 0000000
Then 00000011

The receiver had better know which one we are using!
Binary vs. Unicode

Consider int j = 3;
j holds a binary representation 00...011
We could also write it in Unicode.
The character ‘3’ is coded as 0000000000110011
Binary is better for arithmetic.

The character ‘Ω’ is coded as 0000001110101001
The number 43 can be written as a 32 bit binary integer or as two 16 bit Unicode characters

The receiver had better know which one we are using!
Let’s Examine Three Approaches to external data representation

• CORBA’s Common Data Representation
• Java’s serialization
• Web Service use of XML
CORBA Common Data Representation (CDR) for constructed types

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>length (unsigned long) followed by elements in order</td>
</tr>
<tr>
<td>string</td>
<td>length (unsigned long) followed by characters in order (can also can have wide characters)</td>
</tr>
<tr>
<td>array</td>
<td>array elements in order (no length specified because it is fixed)</td>
</tr>
<tr>
<td>struct</td>
<td>in the order of declaration of the components</td>
</tr>
<tr>
<td>enumerated</td>
<td>unsigned long (the values are specified by the order declared)</td>
</tr>
<tr>
<td>union</td>
<td>type tag followed by the selected member</td>
</tr>
</tbody>
</table>

- Can be used by a variety of programming languages.
- The data is represented in binary form.
- Values are transmitted in sender’s byte ordering which is specified in each message.
- May be used for arguments or return values in RMI.
In CORBA, it is assumed that the sender and receiver have common knowledge of the order and types of the data items to be transmitted in a message.

### CORBA CDR message

<table>
<thead>
<tr>
<th>index in sequence of bytes</th>
<th>4 bytes</th>
<th>notes on representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>5</td>
<td>length of string</td>
</tr>
<tr>
<td>4–7</td>
<td>&quot;Smit&quot;</td>
<td>‘Smith’</td>
</tr>
<tr>
<td>8–11</td>
<td>&quot;h &quot;</td>
<td>length of string</td>
</tr>
<tr>
<td>12–15</td>
<td>6</td>
<td>‘London’</td>
</tr>
<tr>
<td>16–19</td>
<td>&quot;Lond&quot;</td>
<td></td>
</tr>
<tr>
<td>20–23</td>
<td>&quot;on &quot;</td>
<td></td>
</tr>
<tr>
<td>24–27</td>
<td>1934</td>
<td>unsigned long</td>
</tr>
</tbody>
</table>

struct with value: {'Smith', 'London', 1934}
CORBA Interface Definition Language (IDL)

struct Person {
    string name;
    string place;
    long year;
};

corba Interface Compiler

generates

appropriate marshalling and unmarshalling operations
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String nm, place, year) {
        nm = name; this.place = place; this.year = year;
    }
    // more methods
}
Java Serialization

Serialization refers to the activity of flattening an object or even a connected set of objects
- May be used to store an object to disk
- May be used to transmit an object as an argument or return value in Java RMI
- The serialized object holds Class information as well as object instance data
- There is enough class information passed to allow Java to load the appropriate class at runtime. It may not know beforehand what type of object to expect
Java Serialized Form

<table>
<thead>
<tr>
<th>Person</th>
<th>8-byte version number</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>int year</td>
<td>class name, version number</td>
</tr>
<tr>
<td>1934</td>
<td>5 Smith 6 London</td>
<td>number, type and name of instance variables</td>
</tr>
</tbody>
</table>

The true serialized form contains additional type markers; h0 and h1 are handles are references to other locations within the serialized form. The above is a binary representation of {‘Smith’, ‘London’, 1934}.
Web Service use of XML

<p:person p:id="123456789" xmlns:p="http://www.andrew.cmu.edu/~mm6">
    <p:name>Smith</p:name>
    <p:place>London</p:place>
    <p:year>1934</p:year>
</p:person>

- Textual representation is readable by editors like Notepad or Textedit.
- But can represent any information found in binary messages.
- How? Binary data (e.g. pictures and encrypted elements) may be represented in Base64 notation.
- Messages may be constrained by a grammar written in XSD.
- An XSD document may be used to describes the structure and type of the data.
- Interoperable! A wide variety of languages and platforms support the marshalling and un-marshalling of XML messages.
- Verbose but can be compressed.
- Standards and tools still under development in a wide range of domains.
But what about passing pointers?

In systems such as Java RMI or CORBA or .NET remoting, we need a way to pass pointers to remote objects.

Quiz: Why is it not enough to pass along a heap address?
A remote object reference is an identifier for a remote object. May be returned by or passed to a remote method in Java RMI.

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet address</td>
<td>port number</td>
<td>time</td>
<td>object number</td>
<td>interface of remote object</td>
</tr>
</tbody>
</table>

How do these references differ from local references?
A Request Reply Protocol

OK, we know how to pass messages and addresses of objects. But how does the middleware carry out the communication?
UDP Style Request-Reply Communication

d itemView

Client

doOperation

(wait)

(continuation)

Server

getRequest

select object

execute method

sendReply

Request message

Reply message
UDP Based Request-Reply Protocol

**Client side:**

```java
public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
```

sends a request message to the remote object and returns the reply. The arguments specify the remote object, the method to be invoked and the arguments of that method.

**Server side:**

```java
b = getRequest()
operate
sendReply()
```

**Client side**

```java
b = doOperation
```

**Server side**

```java
public byte[] getRequest ();
```
acquires a client request via the server port.

```java
public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
```
sends the reply message reply to the client at its Internet address and port.
Failure Model of UDP Request Reply Protocol

A UDP style doOperation may timeout while waiting.

What should it do?

-- return to caller passing an error message
-- but perhaps the request was received and the response was lost, so, we might write the client to try and try until convinced that the receiver is down

In the case where we retransmit messages the server may receive duplicates
Failure Model Handling Duplicates (Appropriate for UDP but not TCP)

- Suppose the server receives a duplicate messages.
- The protocol may be designed so that either (a) it re-computes the reply (in the case of idempotent operations) or (b) it returns a duplicate reply from its history of previous replies.
- Acknowledgement from client clears the history
Request-Reply Message Structure

<table>
<thead>
<tr>
<th>messageType</th>
<th>int   (0=Request, 1=Reply)</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestId</td>
<td>int</td>
</tr>
<tr>
<td>objectReference</td>
<td>RemoteObjectRef</td>
</tr>
<tr>
<td>methodId</td>
<td>int or Method</td>
</tr>
<tr>
<td>arguments</td>
<td>array of bytes</td>
</tr>
</tbody>
</table>
### RPC Exchange Protocols

Identified by Spector [1982]

<table>
<thead>
<tr>
<th>Name</th>
<th>Messages sent by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Client</td>
</tr>
<tr>
<td>R</td>
<td>Request</td>
</tr>
<tr>
<td>RR</td>
<td>Request</td>
</tr>
<tr>
<td>RRA</td>
<td>Request</td>
</tr>
</tbody>
</table>

R = no response is needed and the client requires no confirmation

RR = a server’s reply message is regarded as an acknowledgement

RRA = Server may discard entries from its history
A Quiz

Why is TCP chosen for request-reply protocols?

Variable size parameter lists.
TCP works hard to ensure that messages are delivered reliably.
So, no need to worry over retransmissions, filtering of duplicates or histories.
The middleware is easier to write.
HTTP Request Message

Traditional HTTP request

<table>
<thead>
<tr>
<th>method</th>
<th>URL or pathname</th>
<th>HTTP version</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>//www.SomeLoc/?age=23</td>
<td>HTTP/ 1.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

HTTP Is Implemented over TCP.
## HTTP SOAP Message

### Web Services style HTTP request

<table>
<thead>
<tr>
<th>method</th>
<th>URL or pathname</th>
<th>HTTP version</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>//SomeSoapLoc/server</td>
<td>HTTP/1.1</td>
<td></td>
<td>&lt;SOAP-ENV</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;age&gt;23...</td>
</tr>
</tbody>
</table>

HTTP is extensible.
Traditional HTTP Reply Message

<table>
<thead>
<tr>
<th>HTTP version</th>
<th>status code</th>
<th>reason</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1</td>
<td>200</td>
<td>OK</td>
<td></td>
<td>&lt;html&gt;…</td>
</tr>
</tbody>
</table>
### HTTP Web Services SOAP Reply Message

<table>
<thead>
<tr>
<th>HTTP version</th>
<th>status code</th>
<th>reason</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1</td>
<td>200</td>
<td>OK</td>
<td></td>
<td>&lt;?xml version..</td>
</tr>
</tbody>
</table>
A Working Toy Example

Server side code:

servant  MyCoolClassServant.java
server   CoolClassServer.java
skeleton MyCool_Skeleton.java
interface MyCoolClass.java

Client side code:

Client   CoolClient.java
Interface MyCoolClass.java
stub     CoolClass_Stub.java

Netbeans 6.8
LowLevelDistributedObjectProject
LowLevelDistributedObjectProjectClient
public class CoolClassServer {

    public static void main(String args[]) {

        System.out.println("Main");

        MyCool_Skeleton cs =
            new MyCool_Skeleton(new MyCoolClass_Servant());

        cs.serve();

    }

}
public class MyCoolClass_Servant implements MyCoolClass {

    private String n[] = {"printer","stereo","TV","ipod","pda"};

    private String a[] = {"HP200XT","Kenwood200","Panasonic","Apple","Palm"};

    public String getDevice(String name) {

        for(int i = 0; i < n.length; i++) {
            if(n[i].equals(name)) return a[i];
        }

        return "No device";
    }
}

public class MyCool_Skeleton {

    MyCoolClass mcc;

    public MyCool_Skeleton(MyCoolClass p) {
        mcc = p;
    }
}
public void serve() {
    try {
        ServerSocket s = new ServerSocket(9000);
        while(true) {
            Socket socket = s.accept();
            ObjectInputStream i = new
                ObjectInputStream(socket.getInputStream());
            String name = (String)i.readObject();
            String result = mcc.getDevice(name);
            ObjectOutputStream o = new
                ObjectOutputStream(socket.getOutputStream());
            o.writeObject(result);
            o.flush();
        }
    } catch(Throwable t) {
        System.out.println("Error " + t);
        System.exit(0);
    }
}
MyCoolClass.java

// Exists on both the client and server

public interface MyCoolClass {
    public String getDevice(String name) throws Exception;
}

public class CoolClient {

    public static void main(String args[]) {
        try {
            MyCoolClass p = new CoolClass_Stub();
            System.out.println(p.getDevice(args[0]));
        }
        catch(Throwable t) {
            t.printStackTrace();
            System.exit(0);
        }
    }
}
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.net.Socket;

public class CoolClass_Stub implements MyCoolClass {

    Socket socket;
    ObjectOutputStream o;
    ObjectInputStream i;
}
CoolClass_Stub.java (2)

```java
public String getDevice(String name) throws Exception {

    socket = new Socket("localhost",9000);
    o = new ObjectOutputStream(socket.getOutputStream());
    o.writeObject(name);
    o.flush();

    i = new ObjectInputStream(socket.getInputStream());

    String ret = (String)(i.readObject());
    socket.close();
    return ret;
}
```
Discussion

With respect to the previous system, let’s discuss:

Request-Reply protocol.
Marshalling and external data representation.
Interoperability.
Security.
Reliability.
Performance.
Openness.
Use of Metadata.
Remote references.
Lab Exercise

Modify the remote object so that it provides an additional method with the signature:

```java
void setDevice(String name, String maker);
```

Modify the client so that it makes use of the new remote method. Show your working solution to your instructor or TA for credit.