

CS 5523 Lecture 9: RMI details and invocation semantics

- *Continue with Java RMI from last lecture*
- *Java object serialization*
- *Callbacks*
- *Design issues for remote calls and invocation*
- *RPCs*
- *Invocation semantics*

Java object serialization:

- *flattens object(s) into compact form for disk storage or message transmission*
- *process doing deserialization has no knowledge of the object structure*

Serialization allows you to save objects to disk and read them back it. Serialization allows you to send objects over a socket or other communication stream and be able to reconstitute a copy on the other end.

Example of a Java object:

```
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;
    }
    // methods
}
```

What do you have to do to serialize this? Ans: Nothing!

Figure 4.9
Indication of Java serialized form

<i>Serialized values</i>				<i>Explanation</i>
Person	8-byte version number		h0	<i>class name, version number</i>
3	int year	java.lang.String name:	java.lang.String place:	<i>number, type and name of instance variables</i>
1934	5 Smith	6 London	h1	<i>values of instance variables</i>

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

Java serialization details:

■ *Serialization file description:*

AC ED (magic number)

00 05 (version number of object serialization format)

■ *Object representation:*

73

class descriptor

object data

■ *Serial numbers:*

■ *class descriptors and objects only appear once in the file*

■ *they are assigned 4-byte serial numbers*

■ *the next time a class or object is encountered, it is specified by the serial number rather than the class description.*

Java serialization details (cont):

■ *Class descriptor:*

72

2-byte length of class name

class name

8-byte fingerprint (based on first 8 bytes of HAS = Secure Hash Algorithm)

1-byte flag (classes that implement Serializable have a flag of 02)

2-byte count of data field descriptors

data field descriptors

78 (end marker)

superclass type or 70 if none

■ *If the same Class is used again in the file:*

71

4-byte serial number

Java serialization details (cont):

■ *Field descriptors:*

1-byte type code: (B = byte, C = char, D = double, ... [= array])

2-byte length of field name

field name

class name (if field is an object)

2-byte count of data field descriptors

data field descriptors

78 (end marker)

superclass type or 70 if none

■ *If the same class is used again in the file:*

71

4-byte serial number

Java serialization details (cont):

■ *Array representation*

75

class descriptor

4-byte number of entries

entries

■ *Other data:*

00

data value

■ *Representation of unicode values uses Universal Transfer Format (UTF)*

Writing your own serialization routines:

Simply implement:

readObject() and writeObject() for the special things.

Java reflection:

- *Reflection is the ability to determine the properties of a class dynamically*
- *The Java package `java.lang.reflect` contains tools for analyzing classes.*
- *Remote object references may be passed as input arguments or returned as output arguments.*

Callbacks:

- *Instead of client polling the server, the server calls a method in the client when it is updated.*
- *Callback refers to server's action in notifying the client*
- *Client creates a remote object that implements an interface for server to call.*
- *Server provides an operation for clients to "register" their callbacks.*
- *When an event occurs, the server calls the interested clients.'*

Callback pluses:

- *More efficient than polling*
- *More timely than polling*
- *Provides a way of server inquiring about client status*

Callback minuses:

- *May leave server with inconsistent state if client crashes or exits without notifying the server*
- *Requires the server to make a series of synchronous RMI's*

Leasing can overcome the first problem. Event notification to address the second problem.

Design issues for remote calls and invocation:

- *What are invocation semantics? (Local calls are invoked exactly once. Under what circumstances can this fail to happen for remote calls?)*
- *Transparency (Local calls are made to in environment of the calling process. How is the choice of environment handled for remote calls?)*

Types of invocation semantics:

- *Exactly once semantics – every method is executed exactly once*
- *Maybe semantics – caller can not determine whether or not the remote method has been executed*
- *At-least-once semantics – caller either receives a result (in which case the user knows the method was executed at least once) or an exception*
- *At-most-once semantics - caller either receives a result (in which case the user knows the method was executed at exactly once) or an exception*

Figure 5.5
Invocation semantics

<i>Fault tolerance measures</i>			<i>Invocation semantics</i>
<i>Retransmit request message</i>	<i>Duplicate filtering</i>	<i>Re-execute procedure or retransmit reply</i>	
No	Not applicable	Not applicable	<i>Maybe</i>
Yes	No	Re-execute procedure	<i>At-least-once</i>
Yes	Yes	Retransmit reply	<i>At-most-once</i>

Figure 5.6
The role of proxy and skeleton in remote method invocation

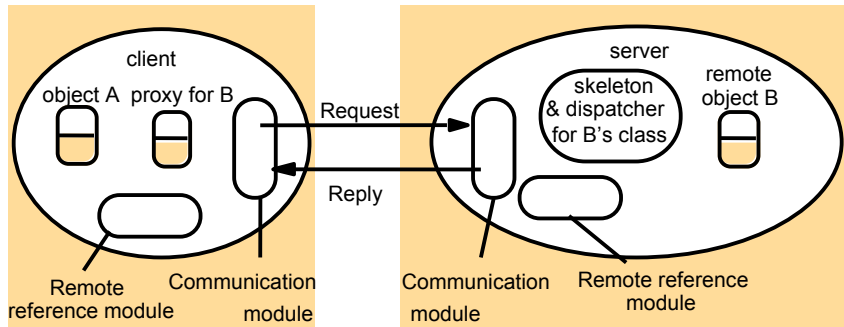
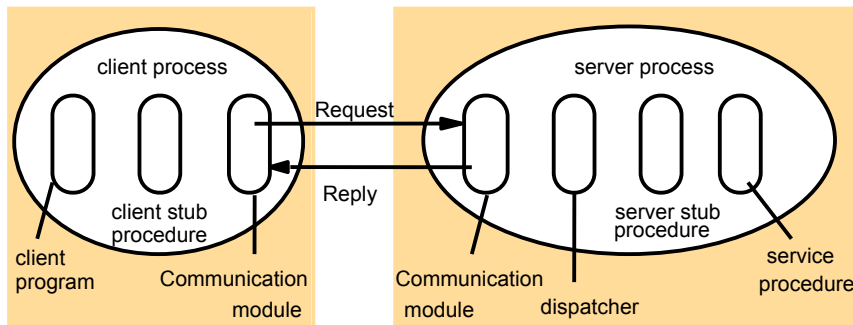


Figure 5.7
Role of client and server stub procedures in RPC



Service interface (RPC):

- A server provides a set of procedures available to client
- These procedures are specified by a service interface
- Input and output parameters are specified
- Use:
 - When the remote procedure is invoked, the values of arguments corresponding to the input parameters are converted to a standard external representation and copied into a packet (marshaling).
 - The client sends the marshaled packet to the server.
 - The server demarshals the packet, performs the procedure, marshals the return packet, and sends the marshaled return packet to the client.
 - Client demarshals the return.
 - The entire procedure is concealed in the call.

Figure 4.14
RPC exchange protocols

<i>Name</i>	<i>Messages sent by</i>		
	<i>Client</i>	<i>Server</i>	<i>Client</i>
R	<i>Request</i>		
RR	<i>Request</i>	<i>Reply</i>	
RRA	<i>Request</i>	<i>Reply</i>	<i>Acknowledge reply</i>

RPC based on TCP or UDP:

RPC can be based on TCP or UDP – what are the design issues with respect to invocation semantics?

Figure 5.8
Files interface in Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
    int length;
    char buffer[MAX];
};
struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};
```

```
struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
    version VERSION {
        void WRITE(writeargs)=1; 1
        Data READ(readargs)=2; 2
    }=2;
} = 9999;
```

Example: Sun RPC:

- *RFC 1831*
- *Used in the Sun NFS network file system*
- *Sometimes called Open Network Computing RPC (ONC RPC)*
- *Can use either UDP or TCP or broadcast UDP.*
- *Uses XDR as an interface definition language*
- *Only single input and output parameters are allowed*
- *Sun RPC runs a local binding services called a port mapper on each host*

For next time:

- *Read CDK 4.3 and 17.1-17.2*