Distributed Systems
COMP90015 2019 SM2
Interprocess Communication
Lectures by Aaron Harwood
© University of Melbourne 2019

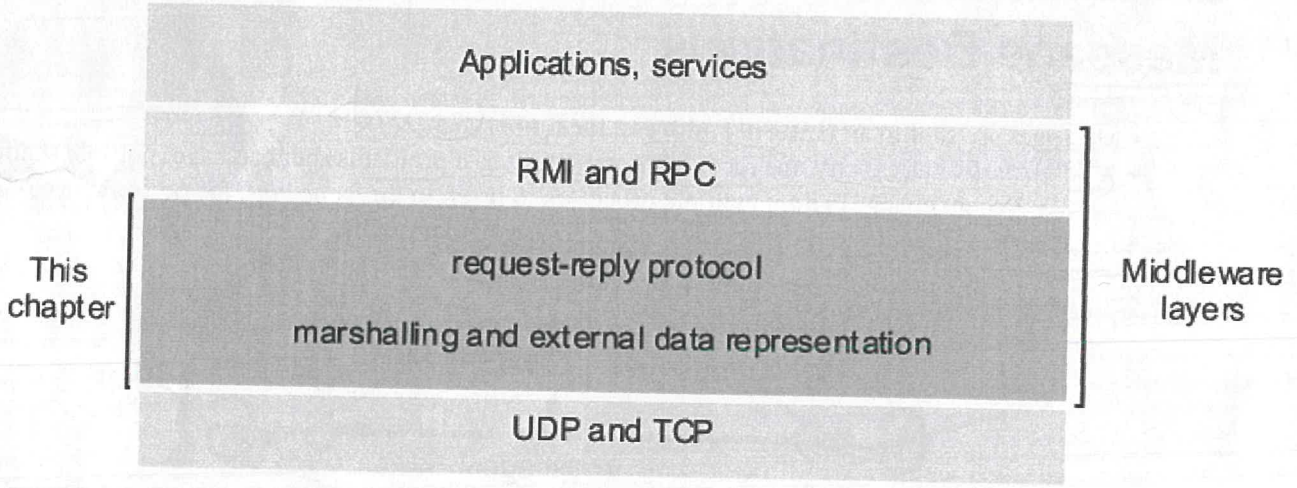
Interprocess Communication Overview

- Introduction
- The API for the Internet protocols
- External data representation and marshalling
- Group communication
- Overlay network

Introduction

This and the next chapters deal with middleware:

- This chapter deals with the lower layer of middleware that support basic interprocess communication
- The next one introduces high level communication paradigms (RMI and RPC)



UDP or User Datagram Protocol, does not guarantee delivery, while TCP or Transport Control Protocols provides a reliable connection oriented protocol.

- Java APIs for Internet protocols:
  - ♦ API for UDP:
    - ◊ Provides a *message passing* abstraction
    - ◊ Is the simplest form of Interprocess Communication (IPC)
    - ◊ Transmits a single message (called a *datagram*) to the receiving process
  - ♦ API for TCP:
    - ◊ Provides an abstraction for a two-way *stream*

◊ Steams do not have message boundaries
◊ Stream provide the basis for producer/consumer communication
◊ Data sent by the producer are queued until the consumer is ready to receive them
◊ The consumer must wait when no data is available

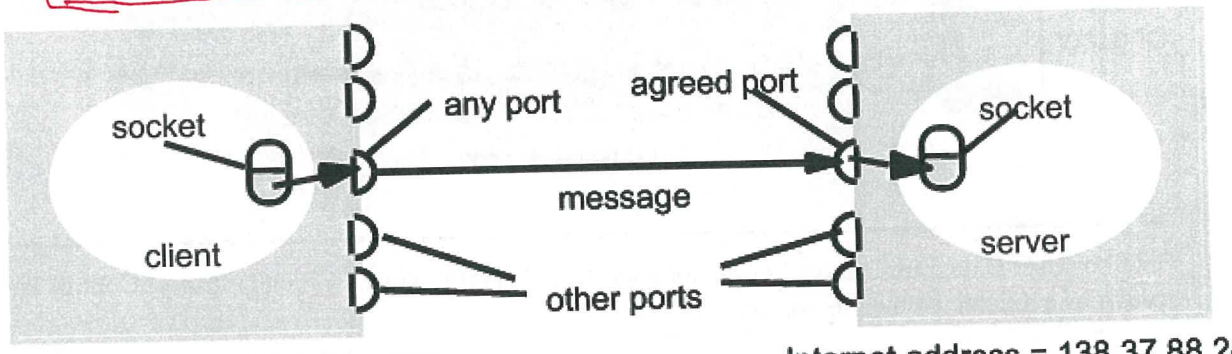
- Data Representation:
  - Deals with how objects and data used in application programs are translated into a form suitable for sending as messages over the network
- Higher level protocols:
  - Client-server communication: Request-reply protocols
  - Group Communication: Group multicast protocol

The API for the Internet protocols

- Processes use two message communication functions: send and receive
- A queue is associated with each message destination
- Communication may be *synchronous* or *asynchronous*
  - In synchronous communication, both *send* and the *receive* operations are *blocking* operations. When a send is issued the sending process is blocked until the receive is issued. Whenever the receive is issued the process blocks until a message arrives.
  - In asynchronous communication, the send operation is non-blocking. The sending process returns as soon as the message is copied to a local buffer and the transmission of the message proceeds in parallel. Receive operation can be blocking or non-blocking (non-blocking receives are not normally supported in today's systems).

Message Destinations

- Messages are sent to an **(Internet address, local port)** pair
- A port *usually* has exactly one receiver (except multicast protocols) but can have multiple senders
  - Recent changes allow multiple processes to listen to the same port, for performance reasons
- Location transparency is provided by a name server, binder or OS



Socket

A **Socket** provides an end point for communication between processes.

- Socket properties:
  - For a process to receive messages, its socket must be bound to a local port on one of the Internet addresses of the computer on which it runs.
  - Messages sent to a particular port of an Internet address can be only be received by a process that has a socket associated with the particular port number on that Internet address.
  - Same socket can be used both for sending and receiving messages.
  - Processes can use multiple ports to receive messages.
  - Ports cannot usually be shared between processes for receiving messages.
    - ◊ Recent changes allow multiple processes to listen on the same port.
  - Any number of processes can send messages to the same port.
  - Each socket is associated with a single protocol (UDP or TCP).

Java Internet Address

- Java provides a class which encapsulates the details regarding Internet Address -- `IntAddress`
- An instance of the `IntAddress` which contains the Internet address can be created by calling the static method `getByName`

```
IntAddress aComputer=IntAddress.getByName("sundowner.cis.unimelb.edu.au");
```

- The method throws UnknownHostException

UDP datagram communication

- Both the sender and the receiver bind to sockets:
  - Server (receiver) binds its socket to a server port, which is made known to the client
  - A client (sender) binds its socket to any free port on the client machine
  - The receive method returns the Internet address and the port of the sender, in addition to the message allowing replies to be sent
- Message Size:
  - Receiving process defines an array of bytes to receive the message
  - If the message is too big it gets truncated
  - Protocol allow packet lengths of 2^16 bytes but the practical limit is 8 kilo bytes.

- Blocking:
  - Non-blocking sends and blocking receives are used for datagram communication
  - Operation returns when the message is copied to the buffer
  - Message is delivered to the message buffer of the socket bound to the destination port
  - Outstanding or future invocations of the receive on the socket can collect the messages
  - Messages are discarded if no socket is bound to the port
- Timeouts:
  - Receive will wait indefinitely till messages are received
  - Timeouts can be set on sockets to exit from infinite waits and check the condition of the sender
- Receive generally allows receiving from any port. It can also allow to receive from only from a given Internet address and port.

- Possible failures:

UDP datagram communication

- Data Corruption: checksum can be used to detect data corruption
- Omission failures: buffers full, corruption, dropping
- Order: messages might be delivered out of order

• UDP does not suffer from overheads associated with guaranteed message delivery
  - Example uses of UDP:
    - Domain Name Service
    - * Voice Over IP (VOIP)

Java API for UDP datagram communication

• Java API provides two classes for datagram communication: `DatagramPacket`, `DatagramSocket`
• `DatagramPacket`
  - supports two constructors, one for creating an instance for sending and the other for creating an instance for receiving
  - other useful methods:
    - `getData()`
    - `getPort()`
    - `getAddress()`

Java API for UDP datagram communication

• `DatagramSocket`
  - supports two constructors, one takes port number and the other with no argument
  - useful methods: `send()`, `receive()`, `setSoTimeout()`, `connect()`

A Java UDP client

```
import  java.net.*;
import java.io.*;

public class UDPClient{
  public static void main(String args[]){
    // args give message contents and destination hostname
    DatagramSocket aSocket = null;
    try {
      aSocket = new DatagramSocket();
      byte [] m = args[0].getBytes();
      InetAddress aHost = InetAddress.getByName(args[1]);
      int serverPort = 6789;
      DatagramPacket request =
      new DatagramPacket(m,  args[0].length(), aHost, serverPort);
      System.out.println("Sending data to server");
      aSocket.send(request);
      byte[] buffer = new byte[1000];
      DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
      System.out.println("Client waiting to receive a response");
      aSocket.receive(reply);
      System.out.println("Reply: " + new String(reply.getData()));
    }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
    }catch (IOException e){System.out.println("IO: " + e.getMessage());
    }finally {if(aSocket != null) aSocket.close();}
```

```
  }
}
```

A Java UDP server

```
import java.net.*;
import java.io.*;

public class UDPServer{
  public static void main(String args[]){

  DatagramSocket aSocket = null;
    try{
    aSocket = new DatagramSocket(6789);
    // create socket at agreed port
    byte[] buffer = new byte[1000];
    while(true){
      DatagramPacket request = new DatagramPacket(buffer, buffer.length);
      System.out.println("Server waiting to receive data");
      aSocket.receive(request);
      System.out.println("Received Data: " + new  String(request.getData()));
      DatagramPacket reply = new DatagramPacket(request.getData(),
            request.getLength(), request.getAddress(), request.getPort());
      aSocket.send(reply);
    }
    }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
    }catch (IOException e) {System.out.println("IO: " + e.getMessage());
    }finally {if(aSocket != null) aSocket.close();}
  }
}
```

TCP Stream Communication

- Features of stream abstraction:
  - **Message sizes**: There is no limit on data size applications can use.
  - **Lost messages**: TCP uses an acknowledgment scheme unlike UDP. If acknowledgments are not received the messages are retransmitted.
  - **Flow control**: TCP protocol attempts to match the speed of the process that reads the message and writes to the stream.
  - **Message duplication or ordering**: Message identifiers are associated with IP packets to enable the recipient to detect and reject duplicates and reorder messages in case messages arrive out of order.
  - **Message destinations**: The communicating processes establish a connection before communicating. The connection involves a connect request from the client to the server followed by an accept request from the server to the client.

- Steps involved in establishing a TCP stream socket:
  - **Client**:
    1. Create a socket specifying the server address and port
    2. Read and write data using the stream associated with the socket
  - **Server**:
    1. Create a listening socket bound to a server port

2. Wait for clients to request a connection (Listening socket maintains a queue of incoming connection requests)
3. Server accepts a connection and creates a new stream socket for the server to communicate with the client retaining the original listening socket at the server port for listening to incoming connections. A pair of sockets in client and server are connected by a pair of streams, one in each direction. A socket has an input stream and an output stream.

- When an application closes a socket, the data in the output buffer is sent to the other end with an indication that the stream is broken. No further communication is possible.
- TCP communication issues:
  - There should a pre-agreed format for the data sent over the socket
  - Blocking is possible at both ends
  - If the process supports threads, it is recommended that a thread is assigned to each connection so that other clients will not be blocked.

- Failure Model:
  - TCP streams use checksum to detect and reject corrupt packets and sequence numbers to detect and reject duplicates
  - Timeouts and retransmission is used to deal with lost packets
  - Under severe congestion TCP streams declare the connections to be broken hence does not provide reliable communication
  - When communication is broken the processes cannot distinguish between network failure and process crash
  - Communicating process cannot definitely say whether the messages sent recently were received
- Use of TCP: HTTP, FTP, Telnet, SMTP

Java API for TCP streams

- Following are the classes used for TCP stream communications:
  - `ServerSocket`:
    - Used to create a socket for listening
    - `accept()` method gets the connect request from the queue and returns an instance of Socket
    - `accept()` blocks until a connections arrives
  - `Socket`:
    - Is used by a pair of processes with a connection
    - Client uses a constructor specifying the DNS hostname and port of the server. This constructor creates the socket associated with a local port and connects it to the remote computer.
    - The methods, `getInputStream()` and `getOutputStream()` provide access to the two data streams.

A Java TCP client

```
import java.net.*;
import java.io.*;
public class TCPClient {
  public static void main (String args[]) {
   Socket s = null;
   try{
     int serverPort = 7899;
     s = new Socket(args[1], serverPort);
     System.out.println("Connection Established");
     DataInputStream in = new DataInputStream( s.getInputStream());
     DataOutputStream out =new DataOutputStream( s.getOutputStream());
     System.out.println("Sending data");
     out.writeUTF(args[0]);        // UTF is a string encoding see Sn. 4.4
     String data = in.readUTF();   // read a line of data from the stream
     System.out.println("Received: "+ data) ;
   }catch (UnknownHostException e) {
      System.out.println("Socket:"+e.getMessage());
   }catch (EOFException e){
      System.out.println("EOF:"+e.getMessage());
   }catch (IOException e){
      System.out.println("readline:"+e.getMessage());
   }finally {
      if(s!=null) try {
        s.close();
      }catch (IOException e){
        System.out.println("close:"+e.getMessage());
      }
   }
   }
}
```

A Java TCP server

corruption . duplicates

```
import java.net.*;
import java.io.*;

public class TCPServer {
  public static void main (String args[]) {
    try{
      int serverPort = 7899; // the server port
      ServerSocket listenSocket = new ServerSocket(serverPort);
      int i = 0;
      while(true) {
        System.out.println("Server listening for a connection");
        Socket clientSocket = listenSocket.accept();
        i++;
        System.out.println("Received connection " + i );
        Connection c = new Connection(clientSocket);
       }
      }
      catch(IOException e)
      {
        System.out.println("Listen socket:"+e.getMessage());
      }
   }
}
```

```
class Connection extends Thread {
```

```
  DataInputStream in;
  DataOutputStream out;
  Socket clientSocket;
  public Connection (Socket aClientSocket) {
    try {
      clientSocket = aClientSocket;
      in = new DataInputStream( clientSocket.getInputStream());
      out =new DataOutputStream( clientSocket.getOutputStream());
      this.start();
    } catch(IOException e) {
       System.out.println("Connection:"+e.getMessage());
    }
  }
  public void run(){
    try {           // an echo server
      System.out.println("server reading data");
      String data = in.readUTF();  // read a line of data from the stream
      System.out.println("server writing data");
      out.writeUTF(data);
   }catch (EOFException e){
      System.out.println("EOF:"+e.getMessage());
   } catch(IOException e) {
      System.out.println("readline:"+e.getMessage());
   } finally{
      try {
        clientSocket.close();
      }catch (IOException e){/*close failed*/}
   }
   }
}
```

External data representation and marshalling

- Data structures in programs are flattened to a sequence of bytes before transmission
- Different computers have different data representations- e.g. number of bytes for an integer, floating point representation, ASCII vs Unicode. Two ways to enable computers to interpret data in different formats:
  - Data is converted to an agreed external format before transmission and converted to the local form on receipt
  - Values transmitted in the senders format, with an indication of the format used
- *External data representation*: Agreed standard for representing data structures and primitive data
- *Marshalling*: Process of converting the data to the form suitable for transmission
- *Unmarshalling*: Process of disassembling the data at the receiver

- Three approaches to external data representation:
  - CORBA's common data representation
  - Java's object serialization
  - Extensible markup language (XML)

Also, JSON is becoming popular, as a lightweight format for communication between servers and web browsers, but now also for general communication between components of a distributed system.

CORBA's Common Data Representation

- CORBA CDR is the external data representation defined with CORBA 2.0
- Consists of 15 primitive data types including short, long, unsigned short, unsigned long, float, double, char, boolean, octet and any
- Primitive data types can be sent in big-endian or little-endian orderings. Values are sent in the sender's ordering which is specified in the message.
- Constructed Types -

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

- Marshalling in CORBA - Marshalling operations can be automatically generated from the data type specification defined in the CORBA IDL (interface definition language). CORBA interface compiler generates the marshalling and unmarshalling operations.

CORBA CDR for a message that contains three fields of a `struct` whose types are `string`, `string` and `unsigned long`:

index in sequence of bytes	← 4 bytes →	notes on representation
0–3	5	length of string
4–7	"Smit"	'Smith'
8–11	"h "	
12–15	6	length of string
16–19	"Lond"	'London'
20-23	"on "	
24–27	1934	unsigned long

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

Java Object Serialization

- Serialization refers to the activity of flattening an object to be suitable for storage or transmission
- Deserialization refers to the activity of restoring the state of the object
- When a Java object is serialized:

- Information about the class of the object is included in the serialization - e.g. name of class, version
- All objects it references are serialized with it. References are serialized as handles (handle is a reference to an object within the serialized object)
- Contents of primitive instance variables that are primitive types (integers, chars etc) are written in a portable format using methods in `ObjectOutputStream` class. Strings and characters are written using a method `writeUTF()` in the same class.

The object `Person p = new Person("Smith", "London", 1934)` in serialized form:

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

handles

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

- During remote method invocation, the arguments and results are serialized and deserialized by middleware.
- *Reflection* property supported by Java allows serialization and deserialization to be carried out automatically.

Extensible markup language (XML)

- A *markup language* is a textual encoding representing data and the details of the structure (or appearance)
- XML is:
  - a markup language defined by World Wide Web Consortium (W3C)
  - tags describe the logical structure of the data
  - is extensible - additional tags can be defined
  - tags are generic - unlike HTML where tags give display instructions
  - self describing unlike CORBA CDR - tags describe the data
  - tags together with *namespaces* allow the tags to be meaningful
  - since data is textual, it can be read by humans and platform independent
  - since data is textual the messages are large causing longer processing and transmission times and more space to store

---

The XML definition of the `Person` structure:

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <!-- a comment -->
</person>
```

XML elements and attributes:

- *Element*
  - consists of data surrounded by tags - e.g. `<name>Smith</name>`
  - elements can be enclosed within elements - e.g. elements with the tag name is enclosed within the elements with tag person. This allows hierarchical representation.
  - an empty tag with no contents is terminated with `/>` - e.g. `<european/>` could be an empty tag included within `<person> </person>`
- *Attributes* - a start tag may optionally contain attributes (names and values) - e.g. `id="12345678"`

An attribute or an element can be used to represent data. If data contains substructures then an element has to be used. Attributes can only represent simple data types.

XML namespaces:

- is a name for a collection of element types and attributes, that is referenced by a URL
- namespace can be specified with an attribute `xmlns` whose value is the URL referring to the file containing the namespace definition - e.g `xmlns:pers = "http://www.cdk4.net/person"`

xmlns

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk4.net/person"
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1934 </pers:year>
</person>
```

XML Schema defines the elements and attributes that can appear in a document.

```
<xsd:schema xmlns:xsd = URL of XML schema definitions >
    <xsd:element name= "person" type ="personType" />
        <xsd:complexType name="personType">
            <xsd:sequence>
                <xsd:element name = "name"  type="xs:string"/>
                <xsd:element name = "place"  type="xs:string"/>
                <xsd:element name = "year"  type="xs:positivelnteger"/>
            </xsd:sequence>
            <xsd:attribute name= "id"  type = "xs:positivelnteger"/>
        </xsd:complexType>
</xsd:schema>
```

JSON

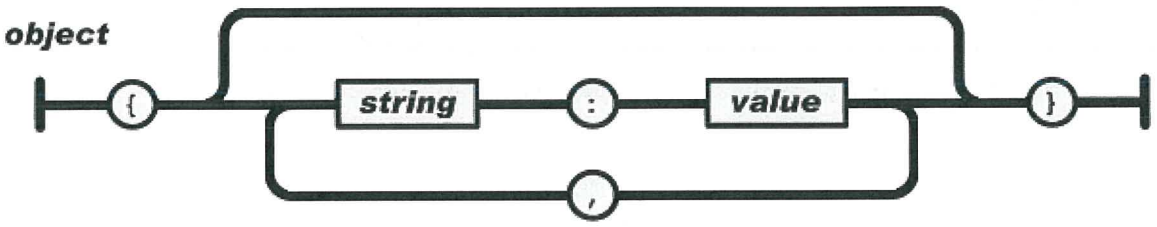
Javascript Object Notation is becoming the dominant format today.

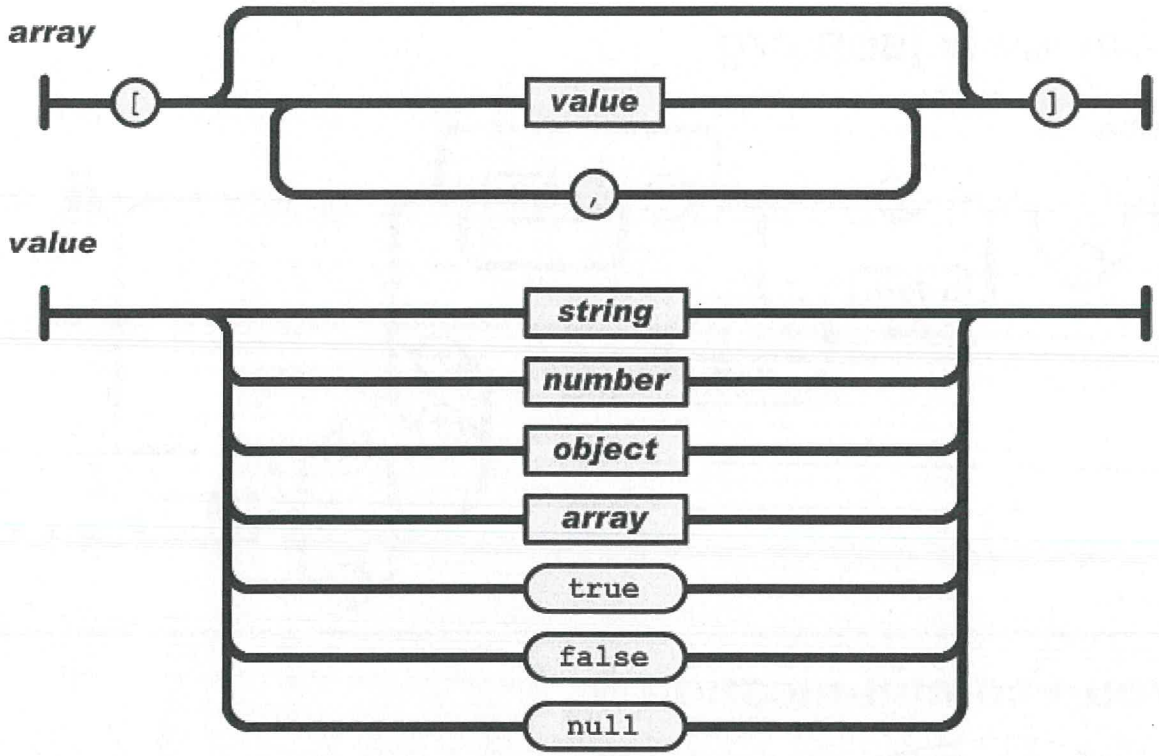
```
{"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]}
```

versus

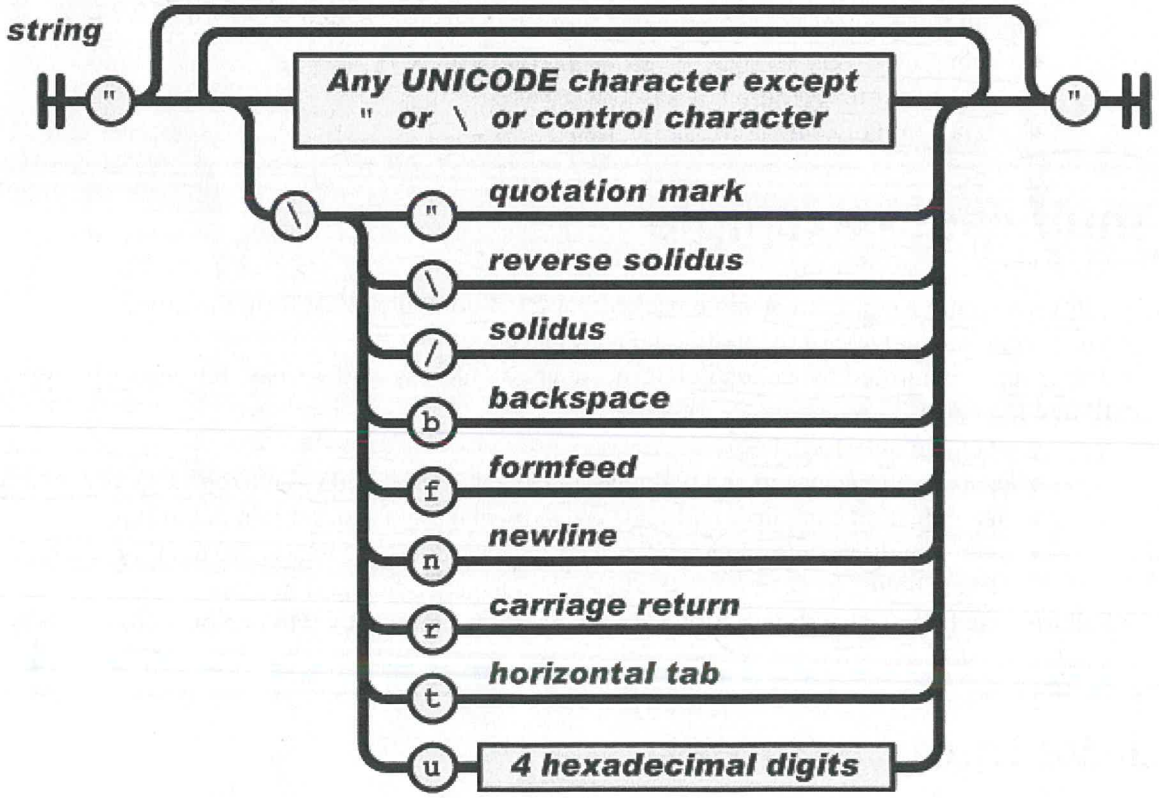
```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
 </employees>
```

From www.json.org

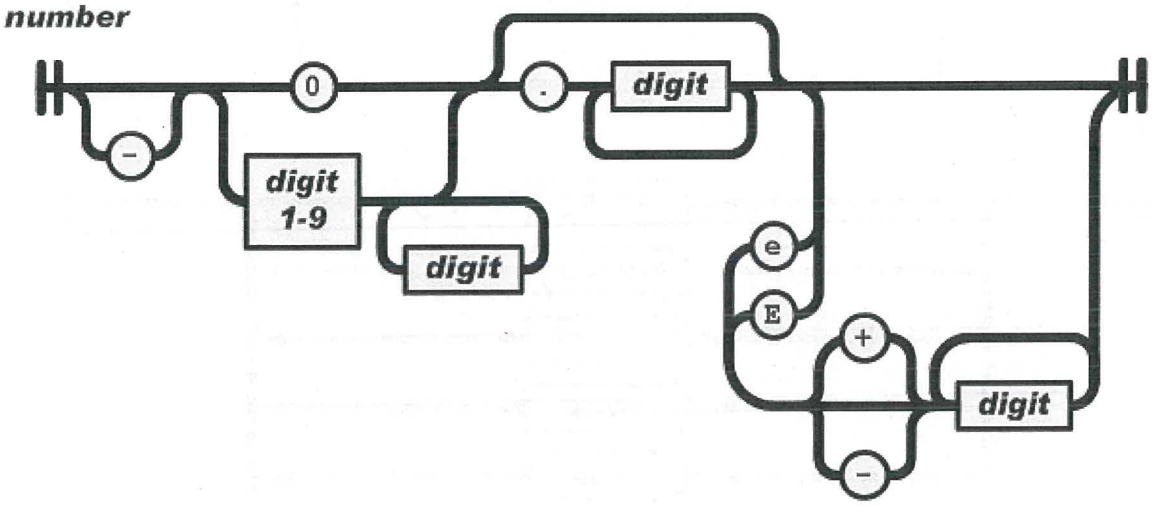




From www.json.org



From www.json.org



Group communication

- A *multicast operation* allows group communication - sending a single message to number of processes identified as a group
- Multicast can happen with or without guarantees of delivery
- Uses of multi cast:
  - Fault tolerance based on replicated services
  - Finding discovery servers
  - Better performance through replicated data
  - propagation of event notification

IP multicast - example

- Allows a sender to transmit a single packet to a set of computers that form the group
- The sender is not aware of the individual recipients
- The group is identified by a class D Internet address (address whose first 4 bits are 1110 in IPv4)
- IP multicast API:
  - available only for UDP
  - an application can send UDP datagrams to a multicast address and ordinary port numbers
  - an application can join a multicast group by making its socket join the group
  - when a multicast message reaches a computer, copies are forwarded to all processes that have sockets bound to the multicast address and the specified port number
- Failure model: Omission failures are possible. Messages may not get to one or more members due to a single omission

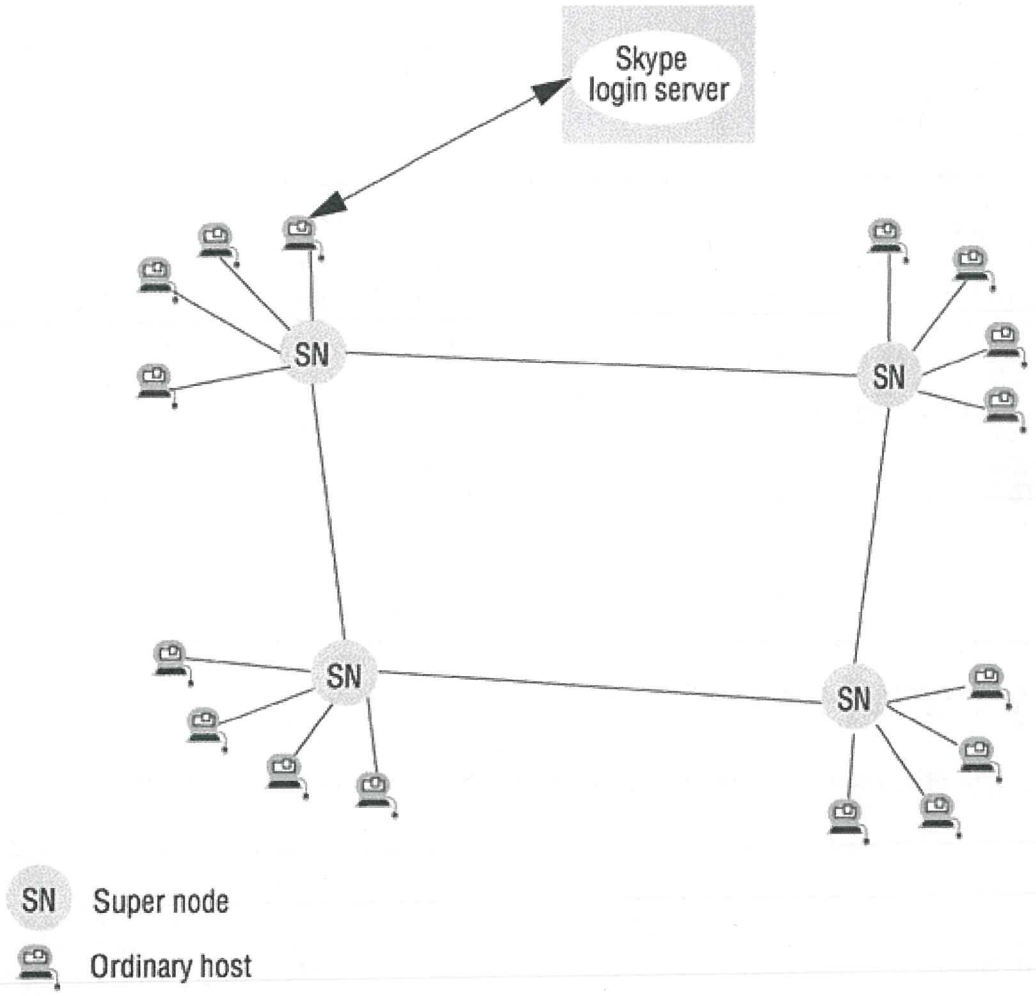
A Java multicast peer

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
  public static void main(String args[]){
    // args give message contents and destination multicast group (e.g. "228.5.6.7")
```

```
    MulticastSocket s =null;
    try {
      InetAddress group = InetAddress.getByName(args[1]);
      s = new MulticastSocket(6789);
      s.joinGroup(group);
      byte [] m = args[0].getBytes();
      DatagramPacket messageOut = new DatagramPacket(m, m.length, group, 6789);
      s.send(messageOut);
      byte[] buffer = new byte[1000];
      for(int i=0; i< 3;i++) {// get messages from others in group
        DatagramPacket messageIn = new DatagramPacket(buffer, buffer.length);
        s.receive(messageIn);
        System.out.println("Received:" + new String(messageIn.getData()));
      }
      s.leaveGroup(group);
    }catch (SocketException e){
      System.out.println("Socket: " + e.getMessage());
    }catch (IOException e){
      System.out.println("IO: " + e.getMessage());
    }finally {if(s != null) s.close();}
  }
}
```

Overlay networks

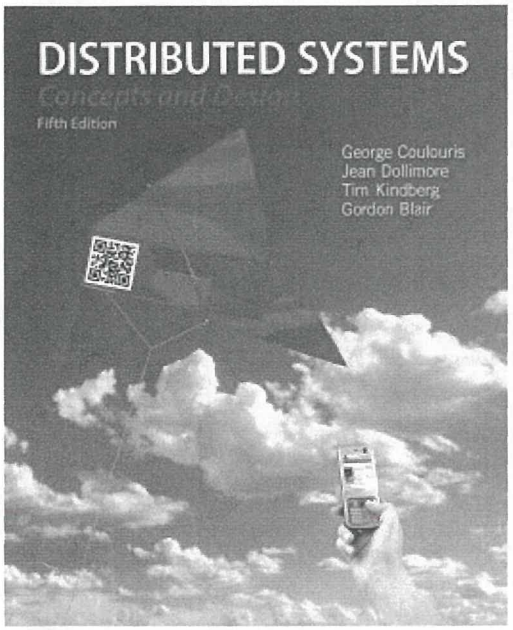
The distributed system forms its own communication network over the Internet, e.g. Skype.


Skype
login server
SN
SN
SN
SN
SN Super node
Ordinary host

Distributed Systems
COMP90015 2019 SM2
Remote Invocation
Lectures by Aaron Harwood
© University of Melbourne 2019

Remote Invocation

From Coulouris, Dollimore and Kindberg, *Distributed Systems: Concepts and Design*, Edition 5, © Addison-Wesley 2012.



Lectures prepared by: Shanika Karunasekera and Aaron Harwood.

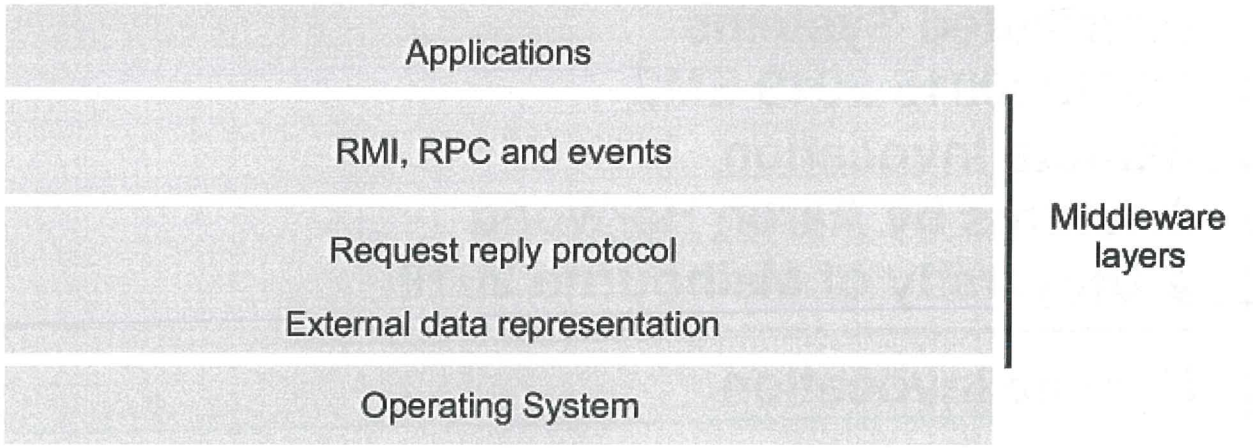
Overview

- Exchange protocols
- Remote Procedure call
- Remote Method Invocation

Introduction

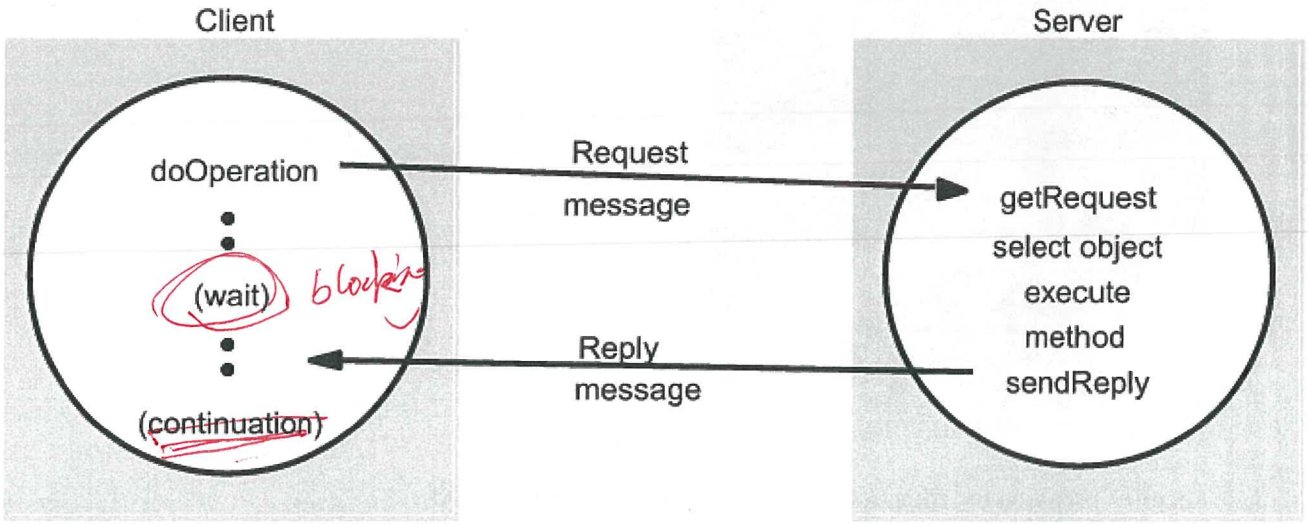
This section covers the high level programming models for distributed systems. Three widely used models are:

- *Remote Procedure Call model* - an extension of the conventional procedure call model.
- *Remote Method Invocation model* - an extension of the object-oriented programming model.



The Request-Reply protocol

The request-reply protocol is perhaps the most common exchange protocol for implementation of remote invocation in a distributed system. We discuss the protocol based on three abstract operations: `doOperation`, `getRequest` and `sendReply`.



Request-Reply Operations

Operations used in the request-reply protocol:

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.

public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message reply to the client at its Internet address and port.

Typical Message Content

A message in a request-reply protocol typically contains a number of fields as shown below.

messageType	int (0=Request, 1=Reply)
requestId	int
objectReference	RemoteObjectRef
methodId	int or Method
arguments	array of bytes

Design Issues

- Failure model can consider:
  - Handling timeouts
  - Discarding duplicate messages
  - Handling lost reply messages - strategy depends on whether the server operations are *idemponent* (an operation that can be performed repeatedly)
  - History - if servers have to send replies without re-execution, a history has to be maintained

Three main design decisions related to implementations of the request/reply protocols are:

- Strategy to retry request message
- Mechanism to filter duplicates
- Strategy for results retransmission

Exchange protocols

- Three different types of protocols are typically used that address the design issues to a varying degree:
  - the request (R) protocol
  - the request-reply (RR) protocol
  - the request-reply-acknowledge reply (RRA) protocol
- Messages passed in these protocols:

Name	Messages sent by		
	Client	Server	Client
R	Request		
RR	Request	Reply	
RRA	Request	Reply	Acknowledge reply

Invocation Semantics

Middleware that implements remote invocation generally provides a certain level of semantics:

- **Maybe invocation semantics**: The remote procedure call may be executed once or not at all. Unless the caller receives a result, it is unknown as to whether the remote procedure was called.
- **At-least-once invocation semantics**: Either the remote procedure was executed at least once, and the caller received a response, or the caller received an exception to indicate the remote procedure was not executed at all.
- **At-most-once**: The remote procedure call was either executed exactly once, in which case the caller received a response, or it was not executed at all and the caller receives an exception.

The level of transparency provided for remote invocation depends on the design choices and objectives. Java Remote Method Invocation supports at-most-once invocation semantics. Sun Remote Procedure Call supports at-least-once semantics.

Fault Tolerance Measures

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

Transparency

Although location and access transparency are goals for remote invocation, in some cases complete transparency is not desirable due to:

- remote invocations being more prone to failure due to network and remote machines
- latency of remote invocations is significantly higher than that of local invocations

Therefore, many implementations provide access transparency but not complete location transparency. This enables the programmer to make optimisation decisions based on location.

Client-server communication

- Client-server communication normally uses the synchronous request-reply communication paradigm
- Involves *send* and *receive* operations
- TCP or UDP can be used - TCP involves additional overheads:
  - redundant acknowledgements
  - needs two additional messages for establishing connection
  - flow control is not needed since the number of arguments and results are limited

HTTP: an example of a RR protocol

- HTTP protocol specifies the:
  - the messages involved in the protocol
  - the methods, arguments and results
  - the rules for marshalling messages
- Allows content negotiation - client specify the data format they can accept
- Allows authentication - based on credentials and challenges
- Original version of the protocol did not persist connections resulting in overloading the server and the network
- HTTP 1.1 uses persistent connections

- HTTP methods

- ♦ GET - Request resources from a URL
- ♦ HEAD - Identical to GET but does not return data
- ♦ POST - Supplies data to the resources
- ♦ PUT - Requests the data to be stored with the given URL
- ♦ DELETE - Requests the server to delete the resource indentified with the given URL
- ♦ OPTIONS - Server supplies the available options
- ♦ TRACE - Server sends back the request message

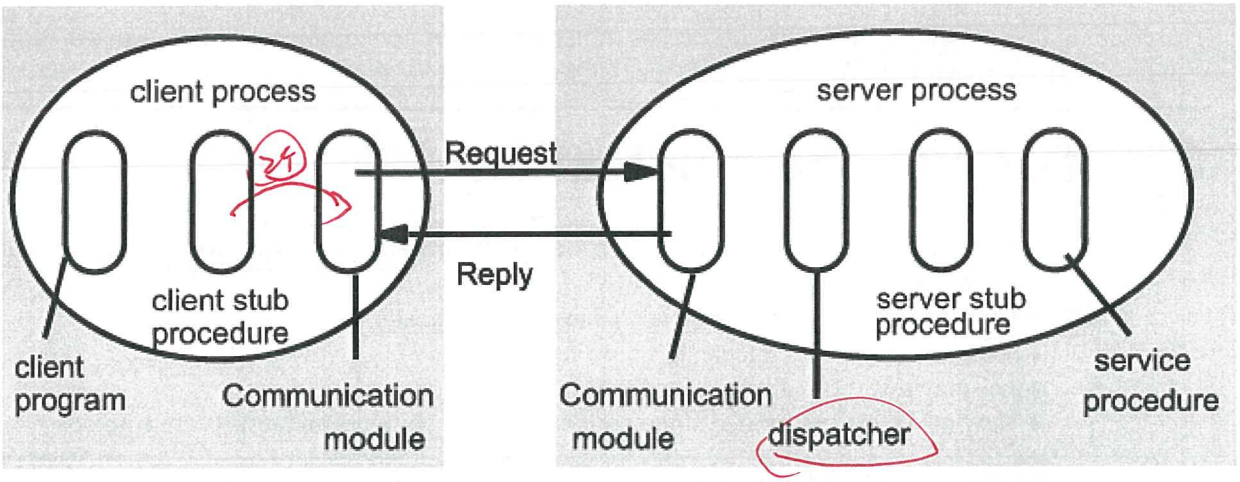
• Requests and replies are marshalled into messages as ASCII text strings

method	URL or pathname	HTTP version	headers	message body
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/1.1		

HTTP version	status code	reason	headers	message body
HTTP/1.1	200	OK		resource data

Remote Procedure Call (RPC)

RPCs enable clients to execute procedures in server processes based on a defined service interface.



Communication Module Implements the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results.

Client Stub Procedure Behaves like a local procedure to the client. Marshals the procedure identifiers and arguments which is handed to the communication module. Unmarshalls the results in the reply.

Dispatcher Selects the server stub based on the procedure identifier and forwards the request to the server stub.

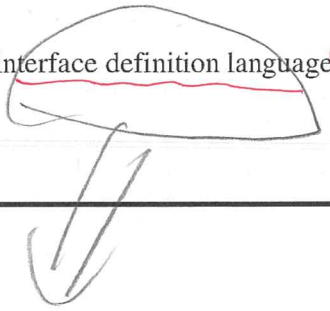
Server stub procedure Unmarshalls the arguments in the request message and forwards it to the service procedure. Marshalls the arguments in the result message and returns it to the client.

Case study: Sun RPC

Reference: UNIX Network programming, W. Richard Stevens, Prentice Hall, Inc.

Sun RPC includes:

- A standard way to define the remote interface using the interface definition language XDR (eXternal Data Representation)
- A compiler `rpcgen` for compiling the remote interface
- A run-time library



Case study: Sun RPC

Step 1: Define the interface using XDR (date.x)

```
/*
 * date.x - Specification of remote date and time service.
 */

/*
 * Define 2 procedures:
 *  bin_date_1() returns the binary time and date (no arguments).
 *  str_date_1() takes a binary time and returns a human-readable string.
 */

program DATE_PROG {
  version DATE_VERS {
    long BIN_DATE(void) = 1;         /* procedure number = 1 */
    string STR_DATE(long) = 2;       /* procedure number = 2 */
  } = 1; /* version number = 1 */
} = 0x31234567;      /* program number = 0x31234567 */
```

Case study: Sun RPC

Step 2: Compile the interface

```
rpcgen date.x
```

This generates the header (date.h), server stub (date*svc.c) and client stub (date*clnt.c).

Case study: Sun RPC

Step 3: Implement the client (rdate.c)

```
/*
 * rdate.c - client program for remote date service.
 */
#include<stdio.h>
#include<rpc/rpc.h>  /* standard RPC include file */
#include"date.h"  /* this file is generated by rpcgen */
```

```
main(argc, argv)
int argc;
char *argv[];
{
  CLIENT  *cl;     /* RPC handle */
  char  *server;
  long  *lresult;  /* return value from bin_date_1() */
  char  **sresult;  /* return value from str_date_1() */
  if (argc != 2) {
     fprintf(stderr, "usage: %s hostname\n", argv[0]);
     exit(1);
  }
  server = argv[1];
```

```
 /*
   * Create the client "handle."
  */
  if ( (cl = clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
   /*
   * Couldn't establish connection with server.
   */
  clnt_pcreateerror(server);
  exit(2);
  }
 /*
 * First call the remote procedure "bin_date".
 */
  if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(3);
  }

  printf("time on host %s = %ld\n", server, *lresult);
```

```
 /*
   * Now call the remote procedure "str_date".
  */
  if ( (sresult = str_date_1(lresult, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(4);
  }
  printf("time on host %s = %s", server, *sresult);
  clnt_destroy(cl);    /* done with the handle */
  exit(0);
}
```

Case study: Sun RPC

Step 4: Implement the server functions (date_proc.c)

```
/*
 * dateproc.c - remote procedures; called by server stub.
 */
#include  <rpc/rpc.h>  /* standard RPC include file */
#include  "date.h"  /* this file is generated by rpcgen */
/*
 * Return the binary date and time.
 */
long *
bin_date_1()
```

```
{
  static long  timeval;  /* must be static */
  long    time();    /* Unix function */
  timeval = time((long *) 0);
  return(&timeval);
}
```

```
/*
 * Convert a binary time and return a human readable string.
 */
char **
str_date_1(bintime)
long  *bintime;
{
  static char  *ptr;    /* must be static */
  char    *ctime();  /* Unix function */
  ptr = ctime(bintime);    /* convert to local time */
  return(&ptr);    /* return the address of pointer */
}
```

Case study: Sun RPC

Step 5: Compile the client and the server

```
gcc -o rdate rdate.c date_clnt.c -lrpcsvc -lnsl

gcc -o date_svc date_proc.c date_svc.c -lrpcsvc -lnsl
```

Case study: Sun RPC

Step 6: Run the server and client

```
./date_svc &

./rdate [hostname]
```

Object-Oriented Concepts

Objects consists of attributes and methods. Objects communicate with other objects by invoking methods, passing arguments and receiving results.

Object References can be used to access objects. Object references can be assigned to variables, passed as arguments and returned as results.

Interfaces define the methods that are available to external objects to invoke. Each method signature specifies the arguments and return values.

Actions - objects performing a particular task on a target object. An action could result in:

- The state of the object changed or queried
- A new object created
- Delegation of tasks to other objects

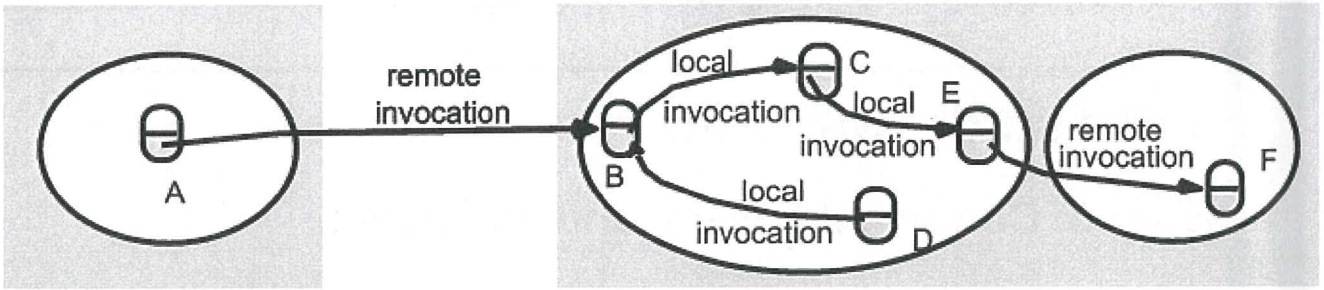
Exceptions are thrown when an error occurs. The calling program catches the exception.

Garbage collection is the process of releasing memory used by objects that are no longer in use. Can be automatic or explicitly done by the program.

Distributed Object Concepts

Remote Objects

An object that can receive remote invocations is called a remote object. A remote object can receive remote invocations as well as local invocations. Remote objects can invoke methods in local objects as well as other remote objects.



Distributed Object Concepts

Remote Object Reference

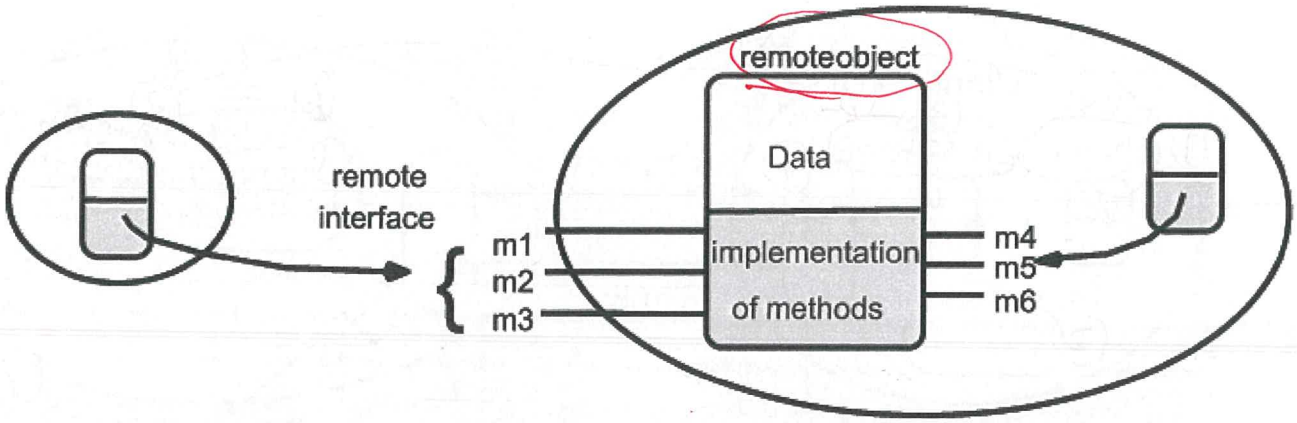
A remote object reference is a unique identifier that can be used throughout the distributed system for identifying an object. This is used for invoking methods in a remote object and can be passed as arguments or returned as results of a remote method invocation.

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

Distributed Object Concepts

Remote Interface

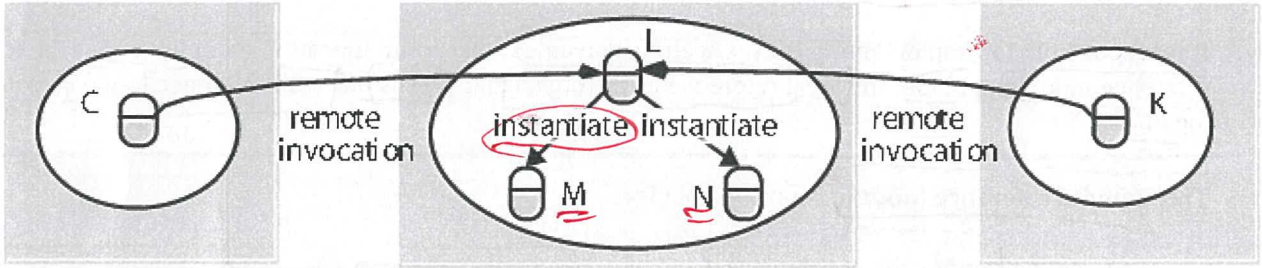
A remote interface defines the methods that can be invoked by external processes. Remote objects implement the remote interface.



Distributed Object Concepts

Actions in a distributed system

Actions can be performed on remote objects (objects in other processes of computers). An action could be executing a remote method defined in the remote interface or creating a new object in the target process. Actions are invoked using Remote Method Invocation (RMI).



Distributed Object Concepts

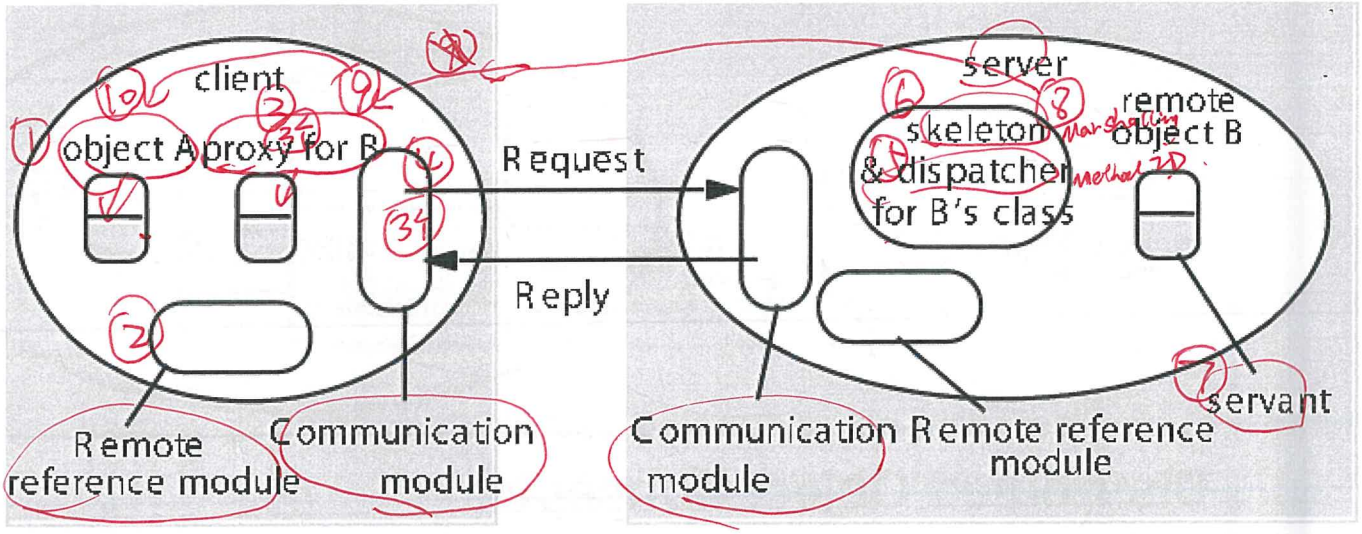
Garbage collection in a distributed system

Is achieved through reference counting.

Exceptions

Similar to local invocations, but special exceptions related to remote invocations are available (e.g. timeouts).

Implementation of RMI



The **Communication Module** is responsible for communicating messages (requests and replies) between the client and the server. It uses three fields from the message:

- message type
- request ID
- remote object reference

It is responsible for implementing the invocation semantics. The communication module queries the remote reference module to obtain the local reference of the object and passes the local reference to the dispatcher for the class.

The **Remote reference module** is responsible for:

- Creating remote object references
- Maintaining the remote object table which is used for translating between local and remote object references

---

The remote object table contains an entry for each:

- Remote object reference held by the process
- Local proxy

Entries are added to the remote object table when:

- A remote object reference is passed for the first time
- When a remote object reference is received and an entry is not present in the table

Servants are the objects in the process that receive the remote invocation.

The RMI software: This is a software layer that lies between the application and the communication and object reference modules. Following are the three main components.

- **Proxy**: Plays the role of a local object to the invoking object. There is a proxy for each remote object which is responsible for:

    - Marshalling the reference of the target object, its own method id and the arguments and forwarding them to the communication module.
    - Unmarshalling the results and forwarding them to the invoking object
- **Dispatcher**: There is one dispatcher for each remote object class. Is responsible for mapping to an appropriate method in the skeleton based on the method ID.
- **Skeleton**: Is responsible for:
    - Unmarshalling the arguments in the request and forwarding them to the servant.
    - Marshalling the results from the servant to be returned to the client.

Developing RMI Programs

Developing a RMI client-server program involves the following steps:

1. *Defining the interface for remote objects* - Interface is defined using the interface definition mechanism supported by the particular RMI software.
2. *Compiling the interface* - Compiling the interface generates the proxy, dispatcher and skeleton classes.
3. *Writing the server program* - The remote object classes are implemented and compiled with the classes for the dispatchers and skeletons. The server is also responsible for creating and initializing the objects and registering them with the binder.
4. *Writing client programs* - Client programs implement invoking code and contain proxies for all remote classes. Uses a binder to lookup for remote objects.

Dynamic invocation

Proxies are precompiled to the program and hence do not allow invocation of remote interfaces not known during compilation.

Dynamic invocation allows the invocation of a generic interface using a `doOperation` method.

Server and Client programs

A server program contains:

- classes for dispatchers and skeletons
- an *initialization* section for creating and initializing at least one of the servants
- code for registering some of the servants with the binder

A client program will contain the classes for all the proxies of remote objects.

Factory methods

- Servants cannot be created by remote invocation on constructors
- Servants are created during initialization or methods in a remote interface designed for this purpose
- **Factory method** is a method used to create servants and a **factory object** is an object with factory patterns

The binder

Client programs require a way to obtain the remote object reference of the remote objects in the server.

A **binder** is a service in a distributed system that supports this functionality.

A binder maintains a table containing mappings from textual names to object references.

Servers register their remote objects (by name) with the binder. Clients look them up by name.

Activation of remote objects

A remote object is *active* if it is available for invocation in the process.

A remote object is *passive* if it is not currently active but can be made active. A passive object contains:

- the implementation of the methods
- its state in marshalled form

Object Location

- Remote object references are used for addressing objects
- The object reference contains the Internet address and the port number of the process that created the remote object
- This restricts the object to reside within the same process
- A **location server** allows clients to locate objects based on the remote object reference

Case Study: Java RMI

Steps to develop a java RMI server:

1. Specify the Remote Interface
2. Implement the Servant Class
3. Compile the Interface and Servant classes
4. Generate skeleton and stub classes
5. Implement the server
6. Compile the server

Steps to develop a java RMI client:

1. Implement the client program
2. Compile the client program

Java RMI Server Example

Specify the Remote Interface

```
import java.util.*;
import java.rmi.*;

public interface RemoteMath extends Remote {
  double add( double i, double j ) throws RemoteException;
  double subtract( double i, double j ) throws RemoteException;
}
```

Java RMI Server Example

Implement the Servant Class

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class RemoteMathServant
       extends UnicastRemoteObject
       implements RemoteMath {
  int _numberOfComputations;

  public RemoteMathServant( ) throws RemoteException {
     _numberOfComputations=0;
  }
  public double add ( double i, double j ) {
    _numberOfComputations++;
    System.out.println("Number of computations performed so far = "
        + _numberOfComputations);
    return (i+j);
  }
  public double subtract ( double i, double j ) {
    _numberOfComputations++;
    System.out.println("Number of computations performed so far = "
        + _numberOfComputations);
    return (i-j);
  }
}
```

Java RMI Server Example

Compile the Interface and Servant classes

```
javac RemoteMath.java
javac RemoteMathServant.java
```

Java RMI Server Example

Generate skeleton and stub classes

```
rmic RemoteMathServant
```

Java RMI Server Example

Implement the server

```
import java.rmi.*;
public class MathServer {
  public static void main(String args[]){
    System.setSecurityManager(new RMISecurityManager());
    System.out.println("Main OK");
    try{
      RemoteMath aMath = new RemoteMathServant();
      System.out.println("After create");
      Naming.rebind("Compute", aMath);
      System.out.println("Math server ready");
    }catch(Exception e) {
      System.out.println("MathServer server main " +
                                          e.getMessage());
    }
  }
}
```

Java RMI Server Example

Compile the server

```
javac MathServer.java
```

Class path should include the classes for the skeleton classes generated by the `rmic` command.

Java RMI Client Example

Implement the client program

```
import java.rmi.*;
import java.rmi.server.*;

public class MathClient {

  public static void main(String[] args) {
    try {
      if(System.getSecurityManager() == null) {
        System.setSecurityManager( new RMISecurityManager() );
      }
      RemoteMath math = (RemoteMath)Naming.lookup("rmi://localhost/Compute");
      System.out.println( "1.7 + 2.8 =  "
                          + math.add(1.7, 2.8) );
      System.out.println( "6.7 - 2.3 =  "
                          + math.subtract(6.7, 2.3) );
    }
    catch( Exception e ) {
      System.out.println( e );
    }
  }
}
```

Compile the client program

```
javac MathClient.java
```

Class path should include the classes for the stub classes generated by the `rmic` command.

Running the Server and Client

1) Develop a security policy file

grant { permission java.security.AllPermission "", ""; };

2) Start RMI registry

rmiregistry &

3) Start server

java -Djava.security.policy=policyfile MathServer

4) Start client

java -Djava.security.policy=policyfile MathClient

Distributed garbage collection

A distributed garbage collector ensures that a remote object continues to exist as long as there are local or remote object references to the object. If no references exist then the object will be removed and the memory will be released.

Java's distributed garbage collection mechanism:

- The server maintains a list of names that hold remote object references to the object `B.holders`
- When a client `A` receives a reference to a remote object `B` it makes a `addRef(B)` invocation to the server which adds `A` to `B.holders`
- When `A`'s garbage collector notices that no references exist for the proxy of remote object `B` it makes a `removeRef(B)` invocation to the server which removes `A` from `B.holders`
- When `B.holders` is empty, the server's local garbage collector releases the space

Note: Race conditions have to be taken care of.