

Distributed Systems & Network Principles

Communication 2/2

Dr Vincent Gramoli | Lecturer
School of Information Technologies



THE UNIVERSITY OF
SYDNEY



› Previous lecture:

- Message-based communication is complex (e.g., routing towards destination, subject to message losses)

› Today's lecture:

- How to avoid message losses?
- How to give the impression that everything happens locally (not remotely)?



- › The Problem of Message Loss
- › The TCP/IP Solution
- › Multithreaded TCP Server in Java
- › Remote Procedure Call (RPC)
- › Remote Method Invocation (RMI)
- › RPC-like Techniques



THE UNIVERSITY OF
SYDNEY

The Problem of Message Loss



Cause

- › Networks are in general **unreliable**
- › Messages can be **lost** (never been delivered even if sent)
- › Examples:
 - A server receives too many requests simultaneously so it cannot treat all
 - A router drops the message because its queue is full

Message losses may impact the computation of a distributed system

Coordinated Attack Problem



- › Constraints of the problem
 - Two armies, each led by a general on separate mountains surrounding a battlefield (distributed system)
 - Can only communicate via messengers (message passing)
 - Messengers can be killed before reaching destination (message losses)
- › Goal: they want to coordinate an attack
 - If they attack at different times, they both die
 - If they attack at the same time, they win

Coordinated Attack Problem (con't)

- › There is no protocols to make sure they will win!



12h!

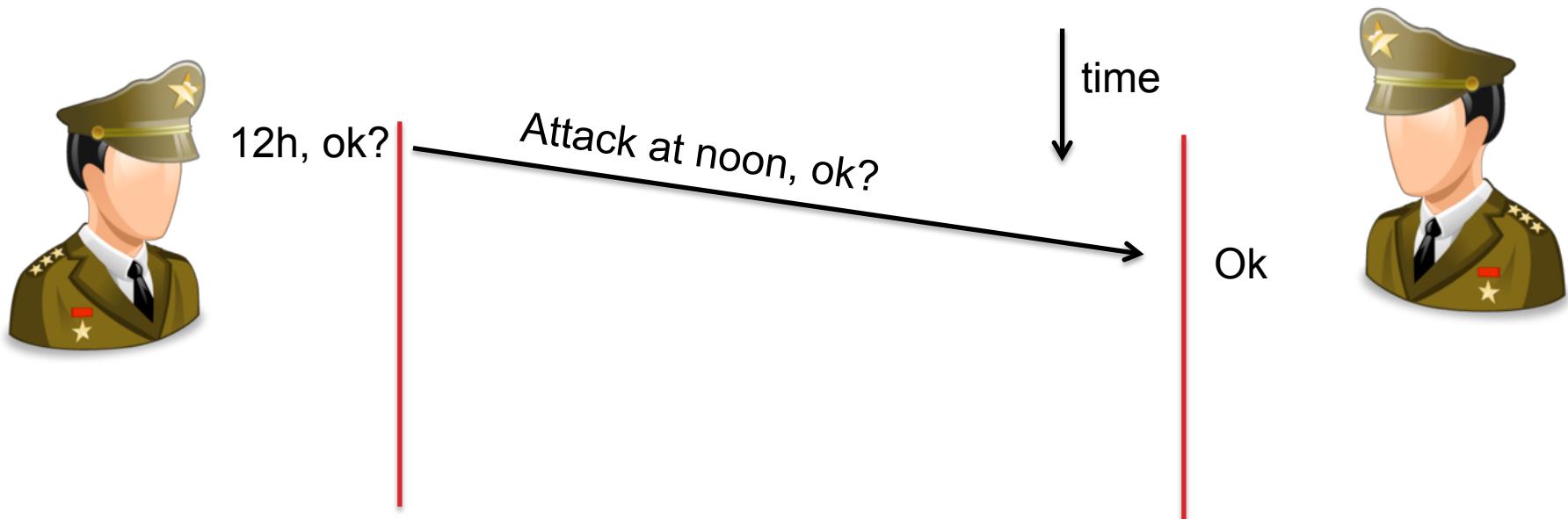


↓
time



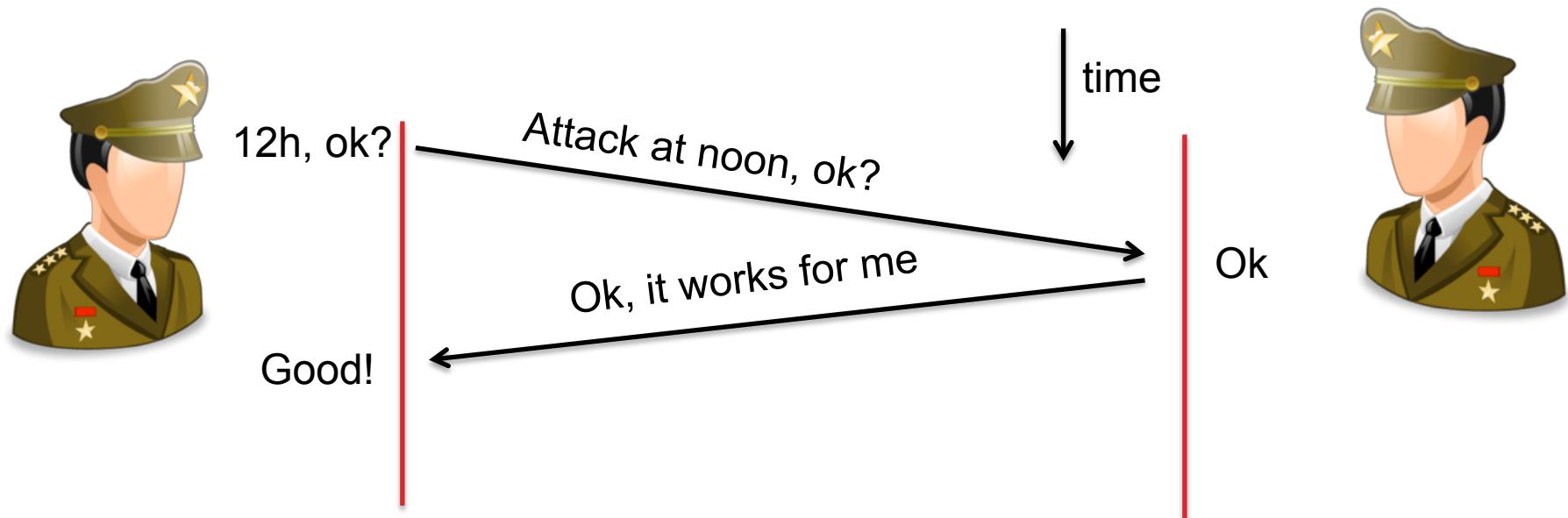
Coordinated Attack Problem (con't)

- › There is no protocols to make sure they will win!



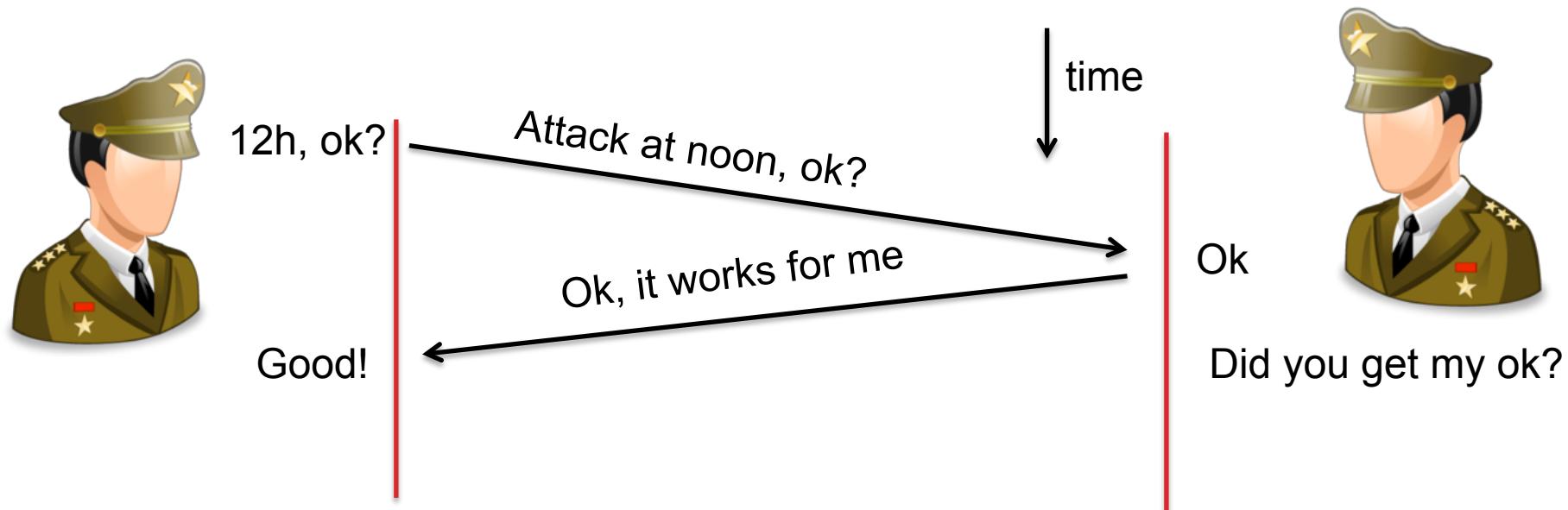
Coordinated Attack Problem (con't)

- › There is no protocols to make sure they will win!



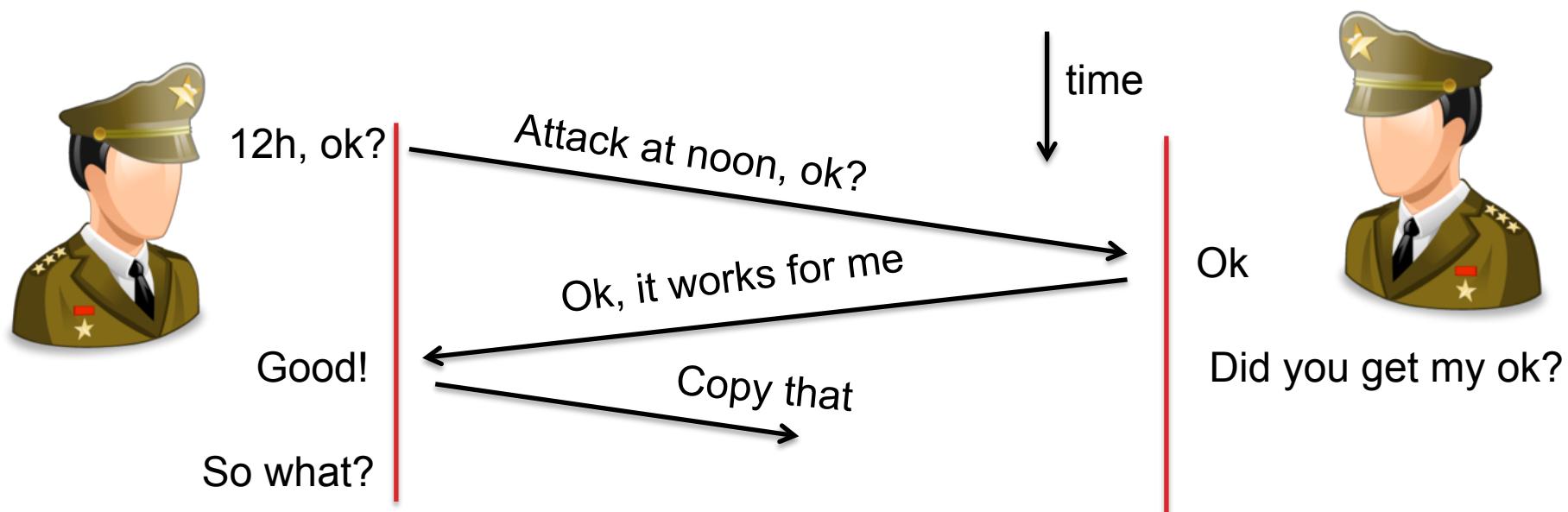
Coordinated Attack Problem (con't)

- › There is no protocols to make sure they will win!



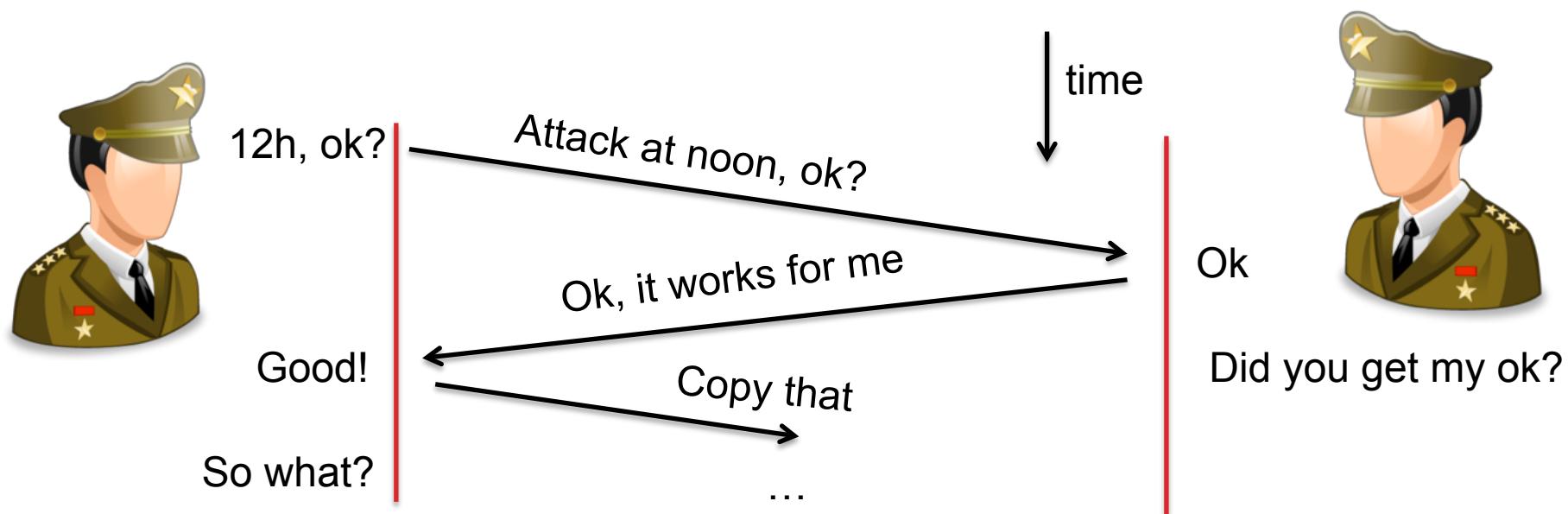
Coordinated Attack Problem (con't)

- › There is no protocols to make sure they will win!



Coordinated Attack Problem (con't)

- › There is no protocols to make sure they will win!



Analogy in networking

- › Constraints of the problem
 - Two remote entities of a distributed system
 - Can only communicate through messages
 - The network is unreliable: messages can be dropped
- › Goal: they want to make sure to do something simultaneously

This is impossible, even if all messages go through

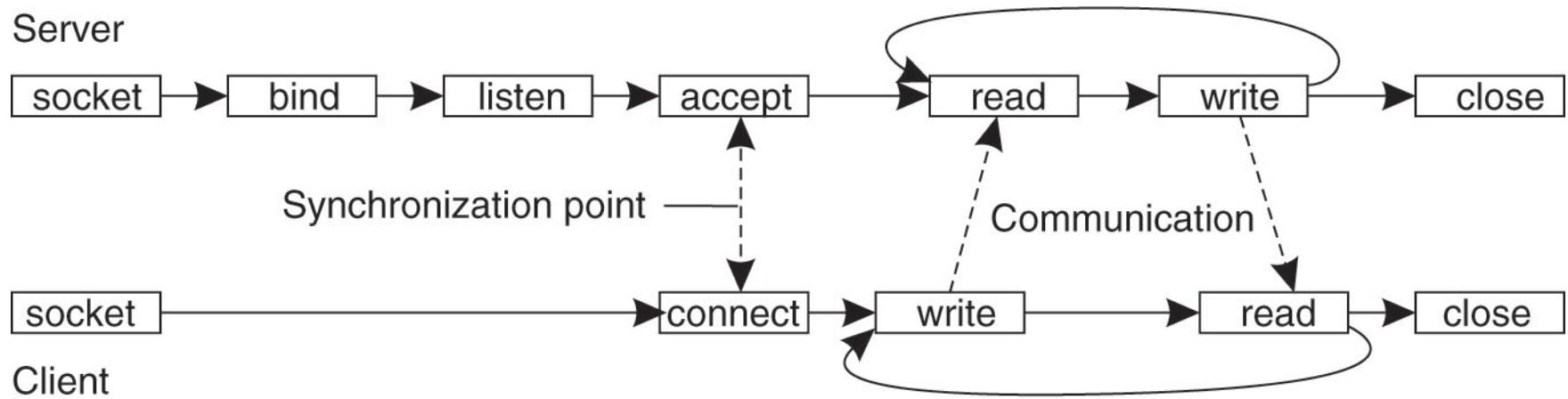


THE UNIVERSITY OF
SYDNEY

The TCP/IP Solution

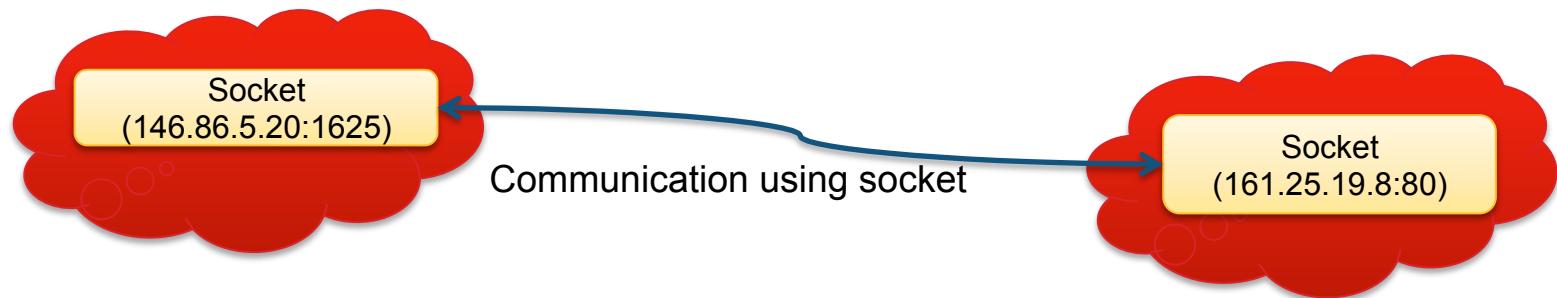
TCP uses sockets

- ## › Recall socket states



TCP is connection-oriented

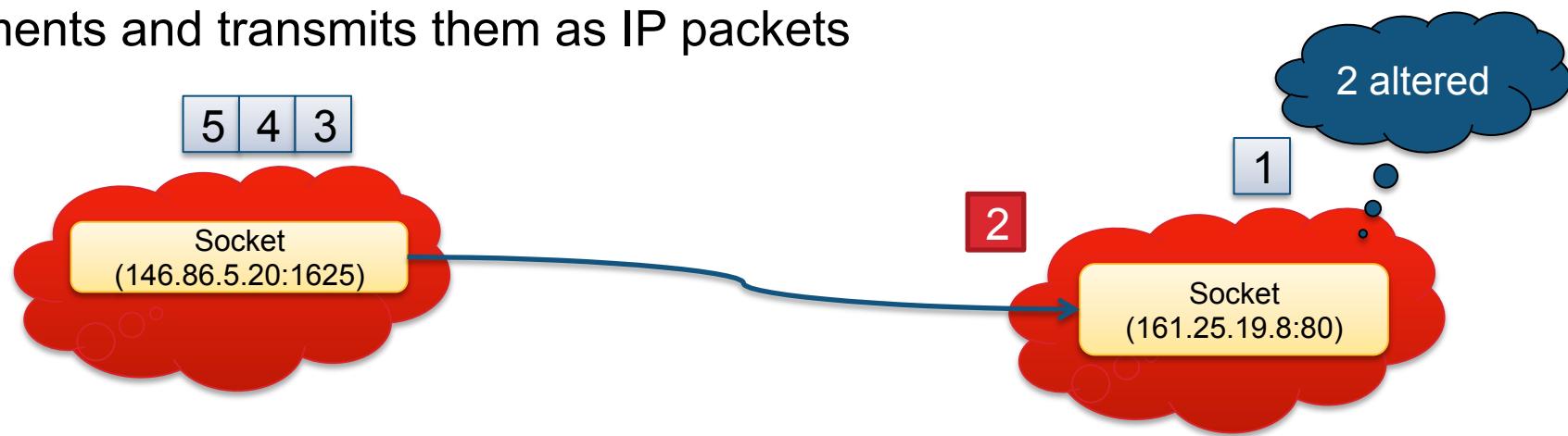
- › Connection: simply an end-to-end agreement to perform reliable data transmission



- › Before any data is transferred the sending and receiving processes must cooperate in the establishment of a bidirectional communication

TCP segments are numbered

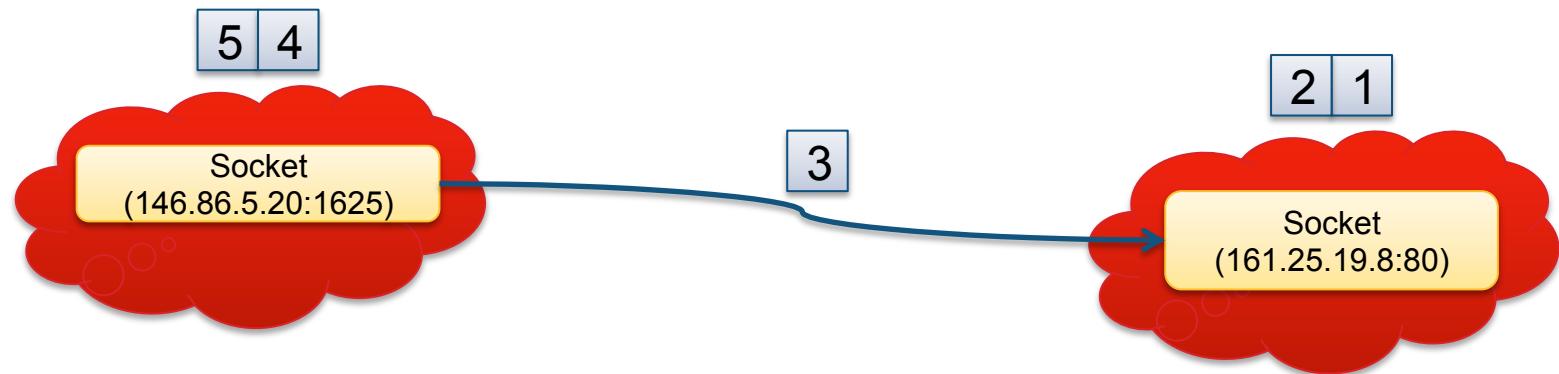
- › A TCP sending process divides the stream into a sequence of data segments and transmits them as IP packets



- › A sequence number is attached to each TCP segment as its first byte
- › A checksum is attached to the segments
- › If a received segments does not match its checksum, the segments is dropped

TCP segments are numbered

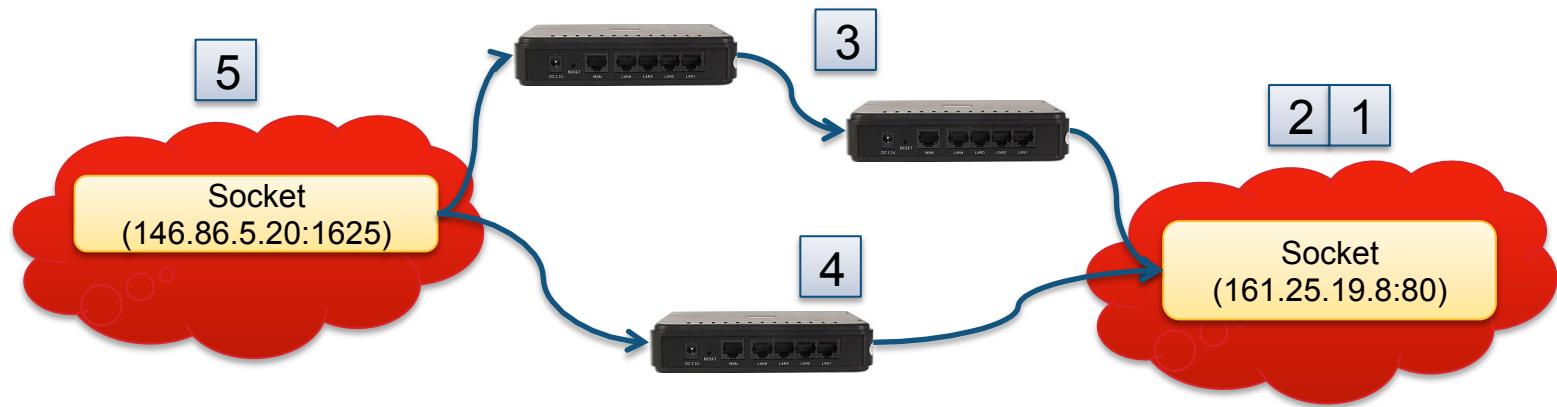
- › A TCP sending process divides the stream into a sequence of data segments and transmit them as IP packets



- › A sequence number is attached to each TCP segment as its first byte

Distinct routes per TCP connection

- › Intermediate nodes have no knowledge of TCP connections

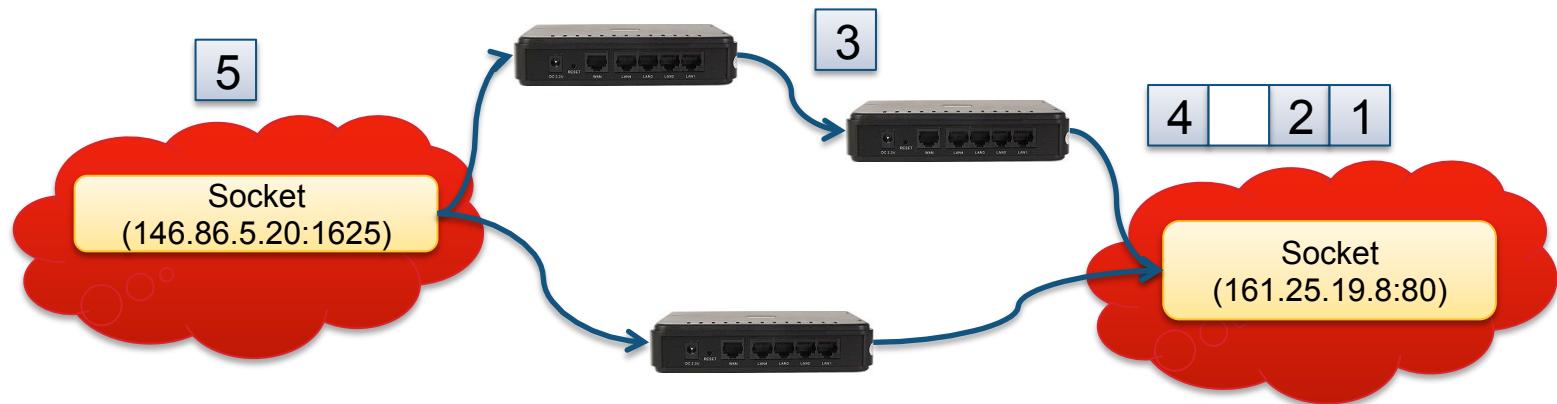


- › The IP packets that transfer the data in a TCP transmission do not necessarily take all the same physical routes



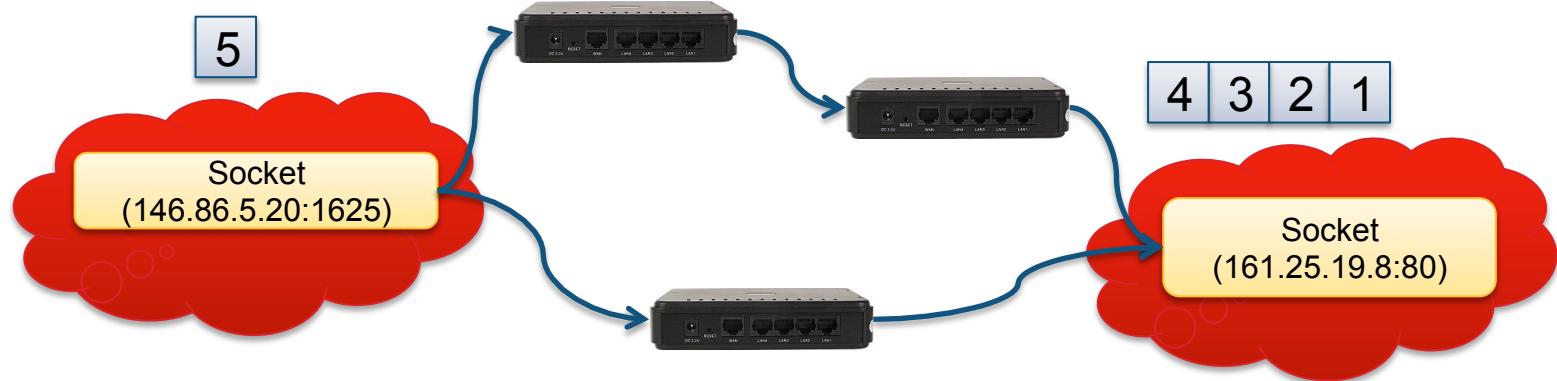
TCP guarantees ordered delivery

- › If segments get re-ordered during the transmission



TCP guarantees ordered delivery (con't)

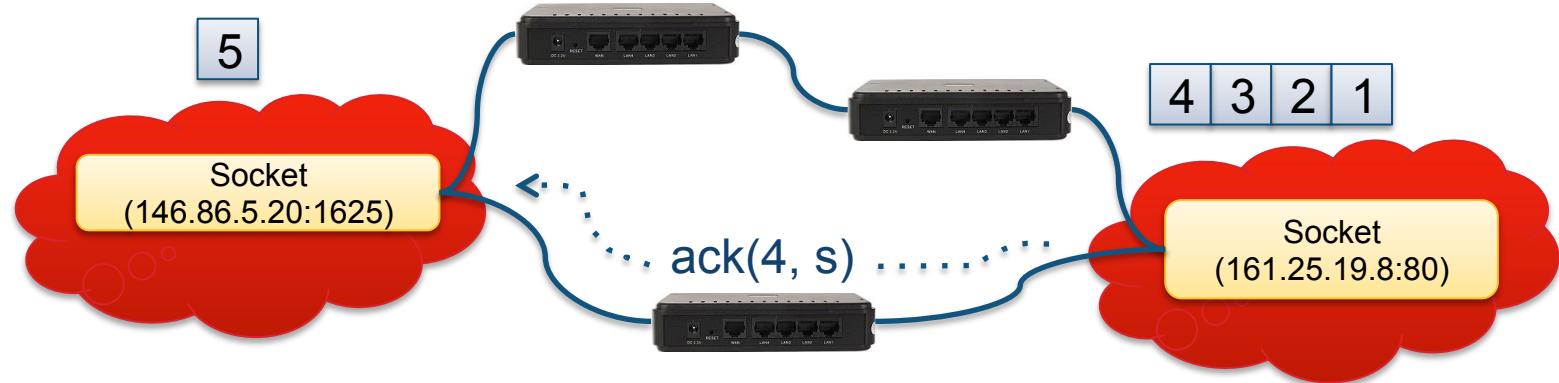
- › If segments get re-ordered during the transmission



- › The receiver exploits the sequence numbers to reorder segments before delivery at its higher layers

TCP controls the flow

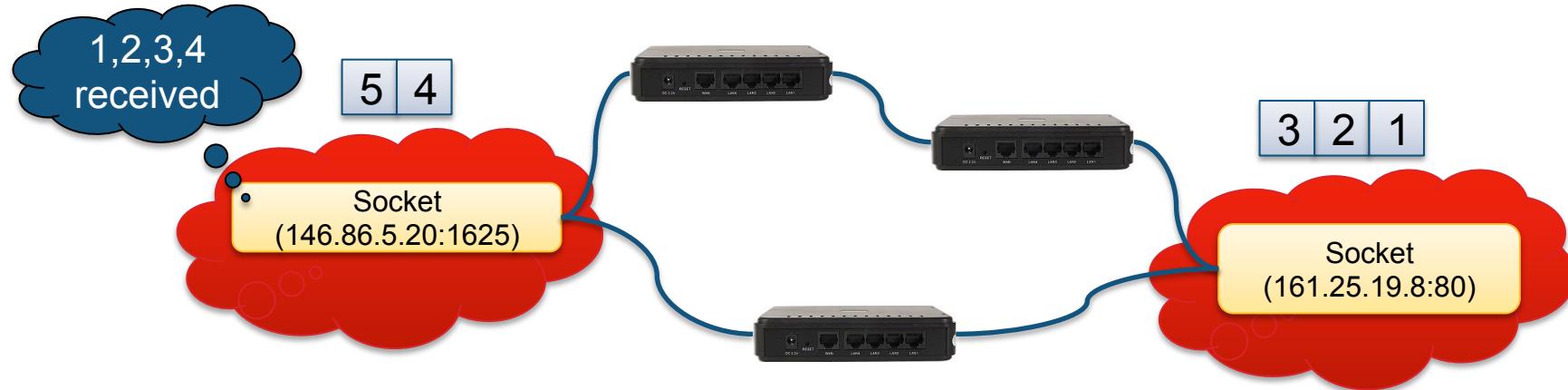
- › The sender takes care not to overwhelm the receiver



- › Whenever the receiver receives a segment it records its number
- › The receiver sends **acknowledgments** to the sender with
 - the highest number seen and
 - a window size, indicating the amount of data the sender can send before waiting for the next acknowledgment

TCP ensures retransmission

- › The sender records the sequence numbers of the segments it has sent



- › Upon reception of an acknowledgment it notes the successful receipts and deletes the appropriate numbers
- › If any segment is not acknowledged within a timeout, then the sender retransmits it



THE UNIVERSITY OF
SYDNEY

Multithreaded TCP Server in Java

TCP in Java

- › A socket class provides a `getInputStream` and a `getOutputStream` for accessing the two streams associated with a socket.
- › The returned abstract classes `InputStream` and `OutputStream` defined the methods for reading and writing bytes
- › Let's implement a **client** that passes data to a **server** that writes it on the standard output
- › We will use `DataOutputStream` and `DataInputStream` to allow binary representations of primitive data types to be read and written in a machine independent manner.

Example: a multithreaded server

- › The client: creates a new socket with specified address and port

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

public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination (e.g., java TCPClient msg 127.0.0.1)
        Socket s = null;
        try {
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream(s.getInputStream()); // for reading from socket
            DataOutputStream out = new DataOutputStream(s.getOutputStream()); // for writing to it
            out.writeUTF(args[0]); // UTF is a string encoding
            String data = in.readUTF();
            System.out.println("Received:" + data);
        } catch(Exception e) { System.out.println(e.getMessage()); }
        finally { if (s!=null) try {s.close();} catch (IOException e) { /* close failed */ } }
    }
}
```

Example: a multithreaded server

- › The client: declares an input stream and an output stream

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

public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination (e.g., java TCPClient msg 127.0.0.1)
        Socket s = null;
        try {
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream(s.getInputStream()); // for reading from socket
            DataOutputStream out = new DataOutputStream(s.getOutputStream()); // for writing to it
            out.writeUTF(args[0]); // UTF is a string encoding
            String data = in.readUTF();
            System.out.println("Received:" + data);
        } catch(Exception e) { System.out.println(e.getMessage()); }
        finally { if (s!=null) try {s.close();} catch (IOException e) { /* close failed */ } }
    }
}
```

Example: a multithreaded server

- › The client: reads (receives) and writes (sends) from these streams

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

public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination (e.g., java TCPClient msg 127.0.0.1)
        Socket s = null;
        try {
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream(s.getInputStream()); // for reading from socket
            DataOutputStream out = new DataOutputStream(s.getOutputStream()); // for writing to it
            out.writeUTF(args[0]); // UTF is a string encoding
            String data = in.readUTF();
            System.out.println("Received:" + data);
        } catch(Exception e) { System.out.println(e.getMessage()); }
        finally { if (s!=null) try {s.close();} catch (IOException e) { /* close failed */ } }
    }
}
```

Example: a multithreaded server

- › The client: catches exception in case of unreachable server or network issue

```
import java.net.*;  
import java.io.*;  
  
public class TCPClient {  
    public static void main (String args[]) {  
        // arguments supply message and hostname of destination (e.g., java TCPClient msg 127.0.0.1)  
        Socket s = null;  
        try {  
            int serverPort = 7896;  
            s = new Socket(args[1], serverPort);  
            DataInputStream in = new DataInputStream(s.getInputStream()); // for reading from socket  
            DataOutputStream out = new DataOutputStream(s.getOutputStream()); // for writing to it  
            out.writeUTF(args[0]); // UTF is a string encoding  
            String data = in.readUTF();  
            System.out.println("Received:" + data);  
        } catch(Exception e) { System.out.println(e.getMessage());}  
        } finally { if (s!=null) try {s.close();} catch (IOException e) { /* close failed */ } }  
    }  
}
```

Example: a multithreaded server

- › The client: in any case closes the connection at the end

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

public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination (e.g., java TCPClient msg 127.0.0.1)
        Socket s = null;
        try {
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream(s.getInputStream()); // for reading from socket
            DataOutputStream out = new DataOutputStream(s.getOutputStream()); // for writing to it
            out.writeUTF(args[0]); // UTF is a string encoding
            String data = in.readUTF();
            System.out.println("Received:" + data);
        } catch(Exception e) { System.out.println(e.getMessage()); }
        } finally { if (s!=null) try {s.close();} catch (IOException e) { /* close failed */ } }
    }
}
```

Example: a multithreaded server (con't)

- › The server: creates a socket with port number (no address required)

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

public class TCPServer {
    public TCPServer() {
        try {
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while (true) {
                Socket clientSocket = listenSocket.accept();          // accept client connection
                Thread t = new Thread(new Connection(clientSocket)); // one thread per connection
                t.start();
            }
        } catch (IOException e) { System.out.println(e.getMessage()); }
    }

    public static void main (String args[]) { TCPServer srv = new TCPServer(); }
}
```

Example: a multithreaded server (con't)

- › The server: spawns a new thread to treat the request concurrently

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

public class TCPServer {
    public TCPServer() {
        try {
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while (true) {
                Socket clientSocket = listenSocket.accept();          // accept client connection
                Thread t = new Thread(new Connection(clientSocket)); // one thread per connection
                t.start();
            }
        } catch (IOException e) { System.out.println(e.getMessage()); }
    }

    public static void main (String args[]) { TCPServer srv = new TCPServer(); }
}
```

Example: a multithreaded server (con't)

- › A thread:
inits streams

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

class Connection implements Runnable {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;

    public Connection(Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream(clientSocket.getInputStream()); // for reading from socket
            out = new DataOutputStream(clientSocket.getOutputStream()); // for writing to it
        } catch (IOException e) { System.out.println(e.getMessage()); }
    }

    public void run() {
        try {
            String data = in.readUTF();
            out.writeUTF(data);
        } catch (Exception e) { System.out.println(e.getMessage()); }
        finally { if (clientSocket!=null) try {clientSocket.close();} catch (IOException e) { /* close failed */ } }
    }
}
```

Example: a multithreaded server (con't)

- › A thread:
reads/writes

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

class Connection implements Runnable {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;

    public Connection(Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream(clientSocket.getInputStream()); // for reading from socket
            out = new DataOutputStream(clientSocket.getOutputStream()); // for writing to it
        } catch (IOException e) { System.out.println(e.getMessage()); }
    }

    public void run() {
        try {
            String data = in.readUTF();
            out.writeUTF(data);
        } catch (Exception e) { System.out.println(e.getMessage()); }
        finally { if (clientSocket!=null) try {clientSocket.close();} catch (IOException e) { /* close failed */ } }
    }
}
```



THE UNIVERSITY OF
SYDNEY

Remote Procedure Call (RPC)

Conventional Procedure Call

- › Call-by-value: a value is simply an initialized local variable. Potential modifications within the called procedure **do not affect the value** outside the procedure scope.
- › Call-by-reference: a reference is a *pointer* (i.e., address) of a variable. If the called procedure modifies the value located at the memory address pointed by the reference, then this **change will be visible** outside the scope of the procedure (e.g., from the caller).
- › Call-by-copy/restore: the variable is copied to the stack by the caller, as in call-by-value, and then copied back after the call, **overwriting the caller's original value**. The semantics may be different from call-by-reference if the same parameter is present multiple times in the parameter list.

Conventional Procedure Call (con't)

- › Example: (a) a combination of **call-by-reference** and **call-by-value** in C
 - Integer i is passed by value
 - Character c is passed by reference

```
void procedure(int i, char *c) {
    i = 1; c[0]='A';
}

int main(int argc, char *argv[]) {
    int i = 0;
    printf("Before call: i=%d, argv[0][0]=%c\n", i, argv[0][0]);
    procedure(i, argv[0]);
    printf("After call: i=%d, argv[0][0]=%c\n", i, argv[0][0]);
    return 0;
}
```

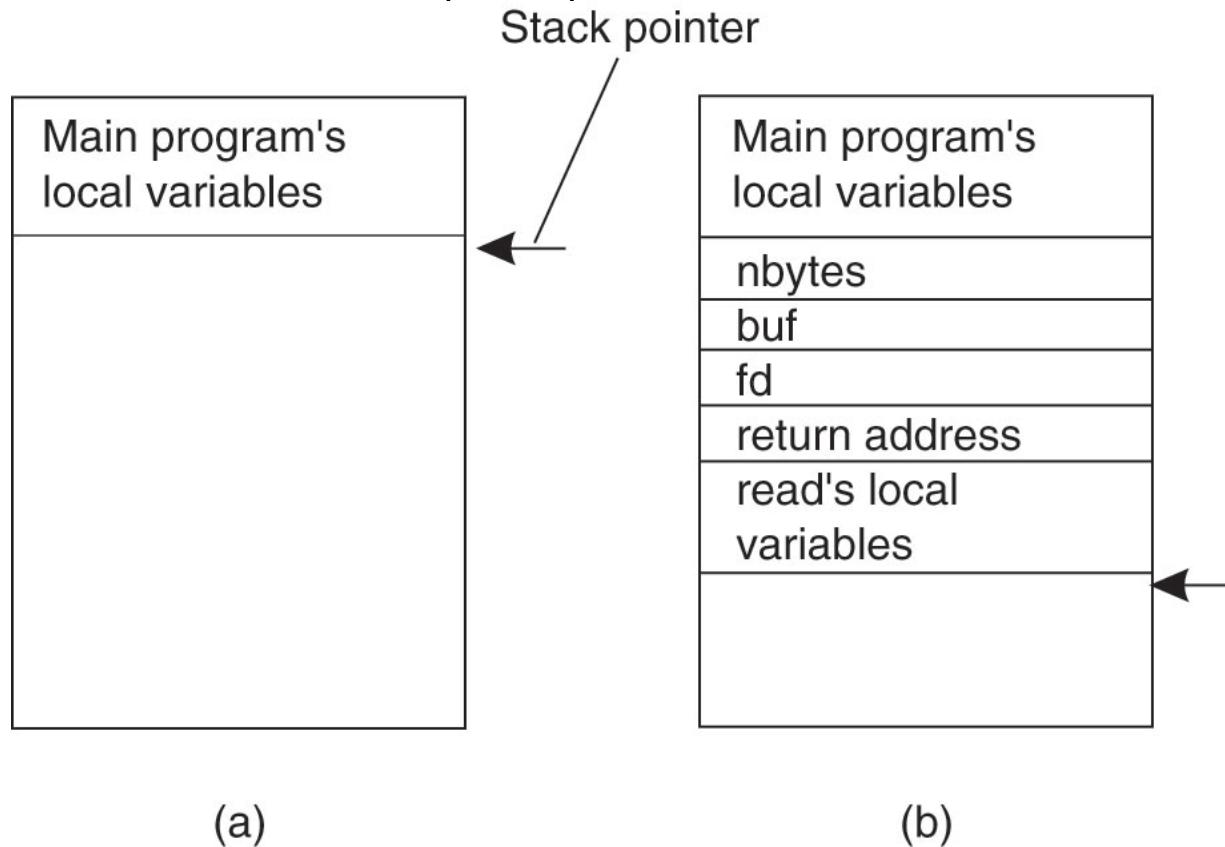
(a) Code of proc.c

```
> gcc -o proc proc.c
> ./proc
Before call: i=0, argv[0][0]=.
After call: i=0, argv[0][0]=A
```

(b) Execution of proc.c

- › (b) **Changes to 'c' are visible outside the procedure, changes to i are not.**

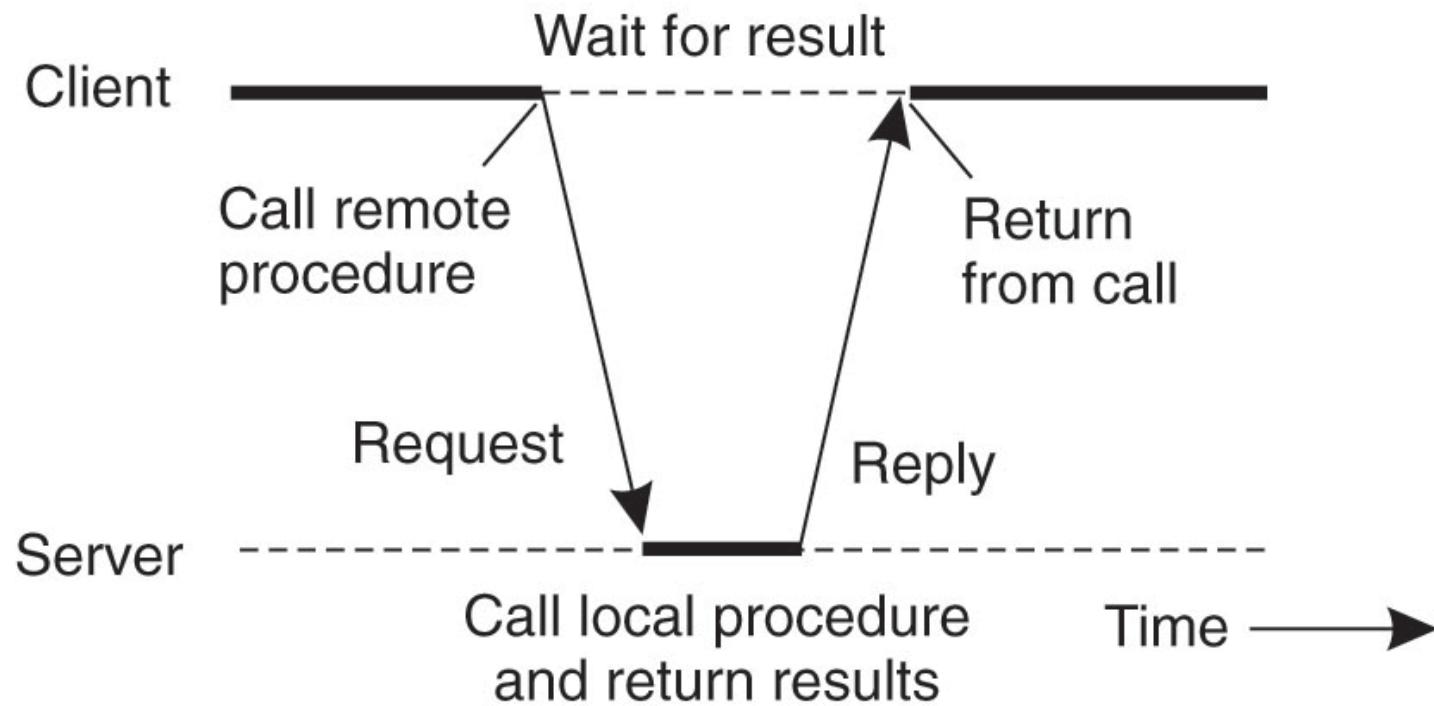
Conventional Procedure Call (con't)



- › (a) The stack before the call to the read procedure
- › (b) The stack while the called procedure is active



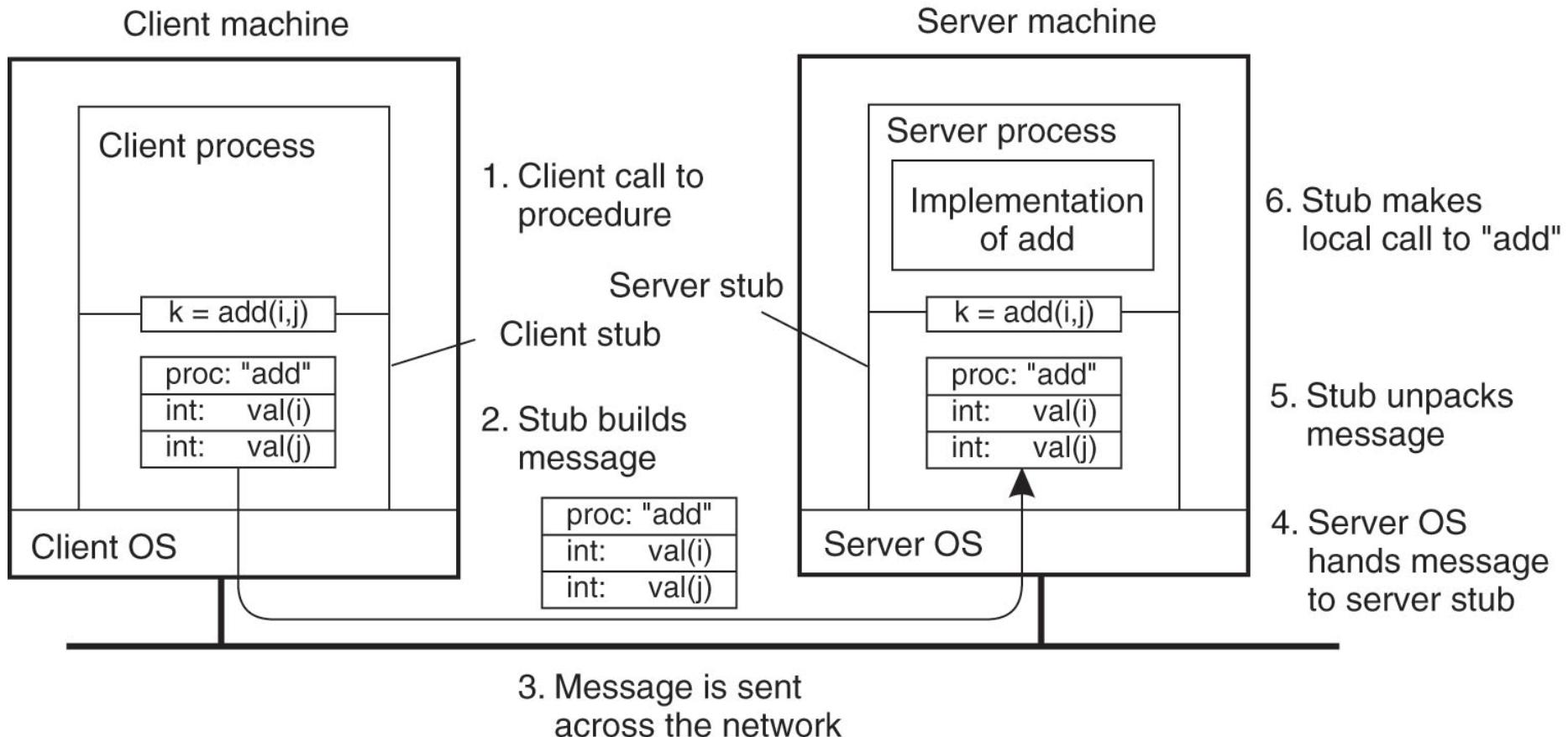
Client and server interaction



A remote procedure call occurs in the following steps:

1. The client procedure **calls the client stub** in the normal way.
2. The client stub **builds a message** and calls the local operating system.
3. The client's OS **sends the message** to the remote OS.
4. The remote OS **gives the message** to the server stub.
5. The server stub **unpacks the parameters** and calls the server.
6. The server does the work and **returns the result** to the stub.
7. The server stub **packs it** in a message and calls its local OS.
8. The server's OS **sends the message** to the client's OS.
9. The client's OS **gives the message** to the client stub.
10. The stub **unpacks the result** and returns to the client.

Remote procedure call example: add(int i, int j) which sums up 2 integers, $i+j$.



Parameter marshalling: packing parameters into a message.

1. The **client** stub **marshals** the following into a message before sending it:
 - The two parameters, i and j,
 - The identifier (name or number) of the procedure ‘add’ to be called.
2. The **server** stub **unmarshals** the message and calls the procedure on the server
3. Once done, the **server** stub **marshals** the result $i+j$ to send it back
4. The **client** stub **unmarshals** this message and returns to the client

Why the type (e.g., integer, string) information must be sent

- › Different architectures may have different data representations:
 - SPARC (<v9) uses the *big endian* format to number bytes (numbered from left to right)
 - x86 uses the *little endian* format to number bytes (numbered from right to left)

3	2	1	0
0	0	0	5
7	6	5	4
L	L	I	J

0	1	2	3
5	0	0	0
4	5	6	7
J	I	L	L

0	1	2	3
0	0	0	5
4	5	6	7
L	L	I	J

(a)

(b)

(c)

- › (a) original message sent from the Intel Pentium (x86) reads 5 and “JILL”
- › (b) message after reception on the SPARC: it reads $5 \cdot 2^{24}$ and “JILL”
- › (c) message after inversion is still incorrect: it reads 5 and “LLIJ”
- › Hence the need for specifying types (integer or string).

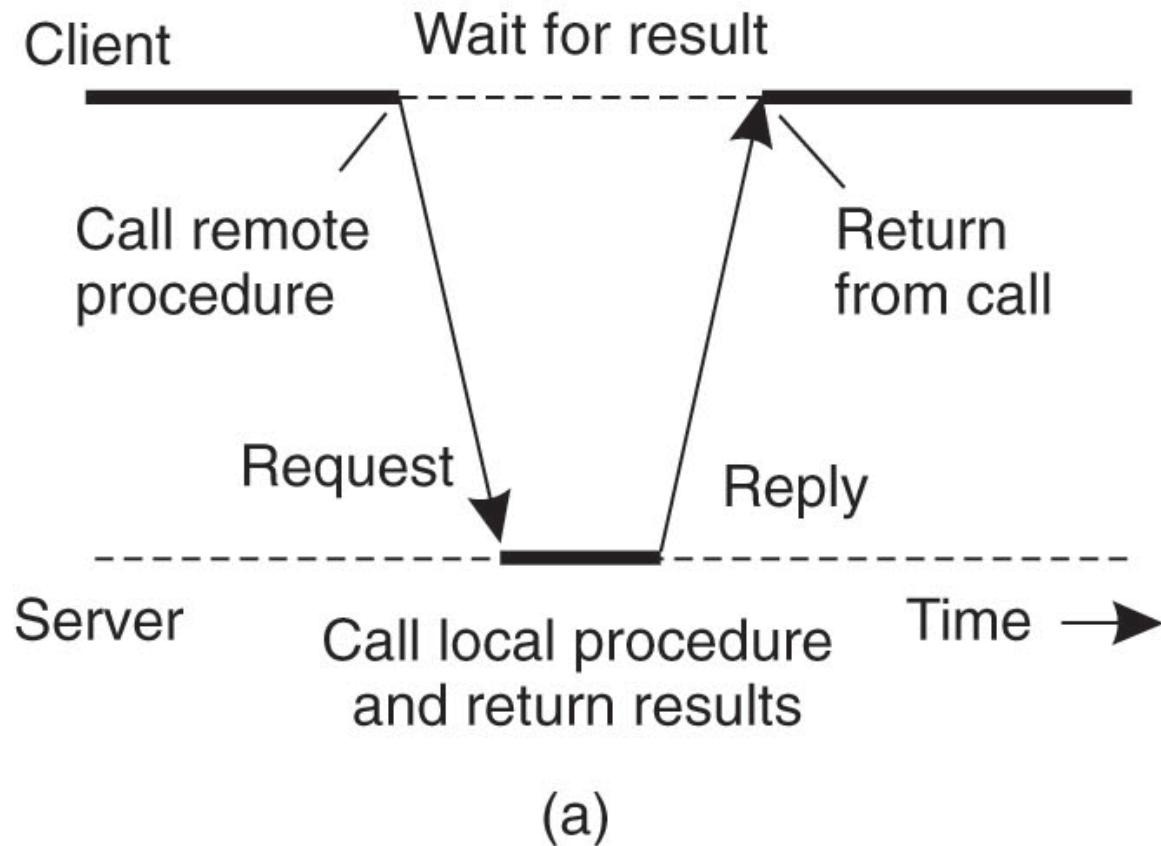
Passing reference parameters in a RPC

- › Reference is only **meaningful** within a given **address space**
- › Different processes have **different address spaces**
 - on the **client**, address 1000 may points the argv[0][0] while
 - on the **server**, address 1000 may point to somewhere in the program text
- › The solution is to **call-by-copy/restore** instead of **call-by-reference**
 - The client stub marshalls the **pointer and the value it points to** into the message
 - The server stub unmarshalls the message **determines the pointer** that points to this value (which may be different) and call the procedure on the server
 - If the server modified the value located at this pointer, then the **modifications are visible from the stub**
 - Once the result message is sent back to the client stub, it is copied to the client



Synchronous RPC

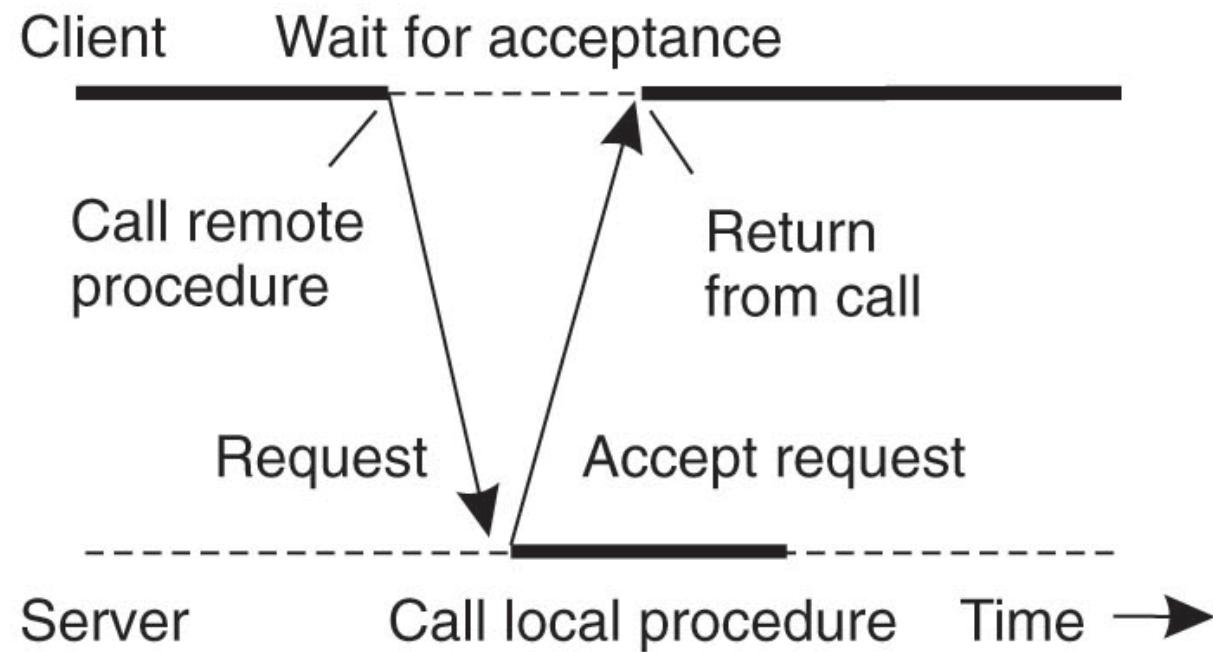
- › Interaction between client and server in a normal (**synchronous**) RPC





Asynchronous RPC

- › Interaction with an **asynchronous** RPC: e.g., when no result is returned

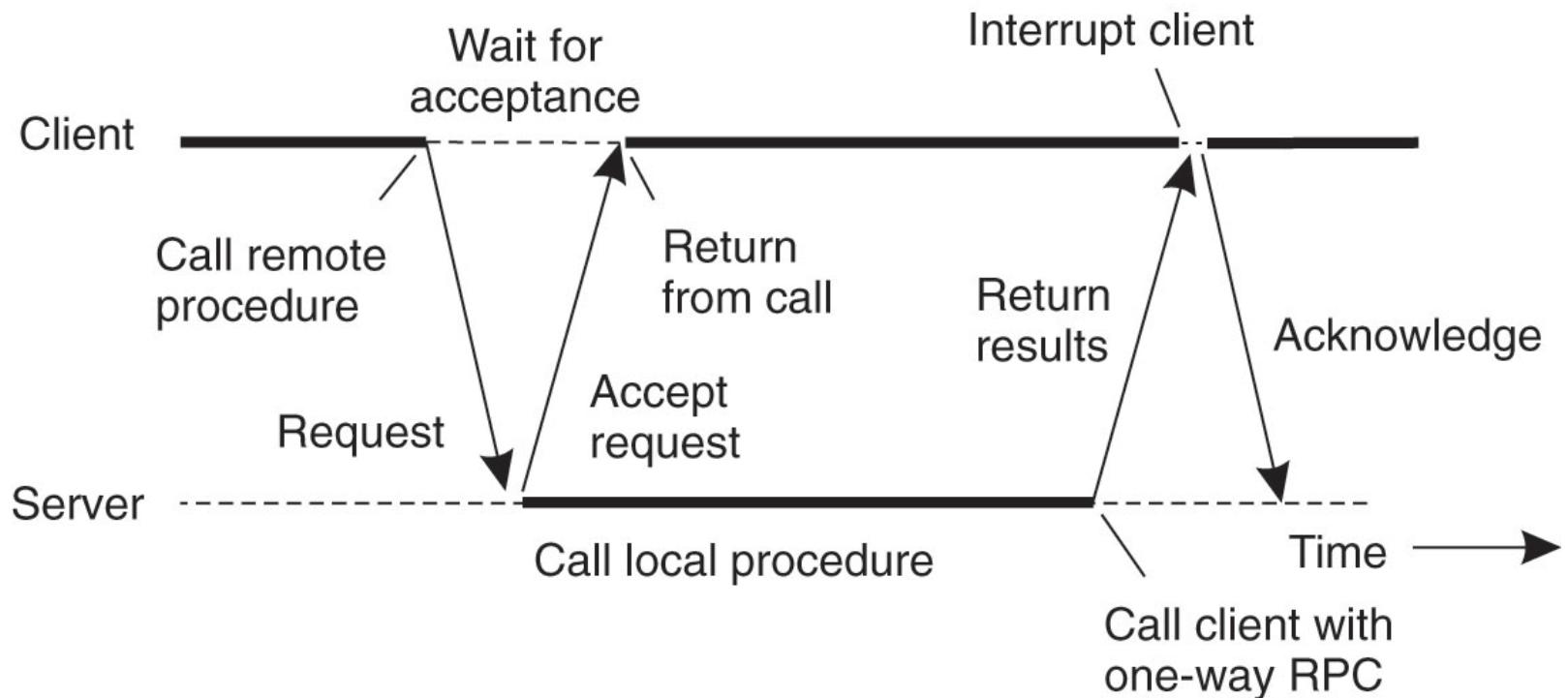


(b)



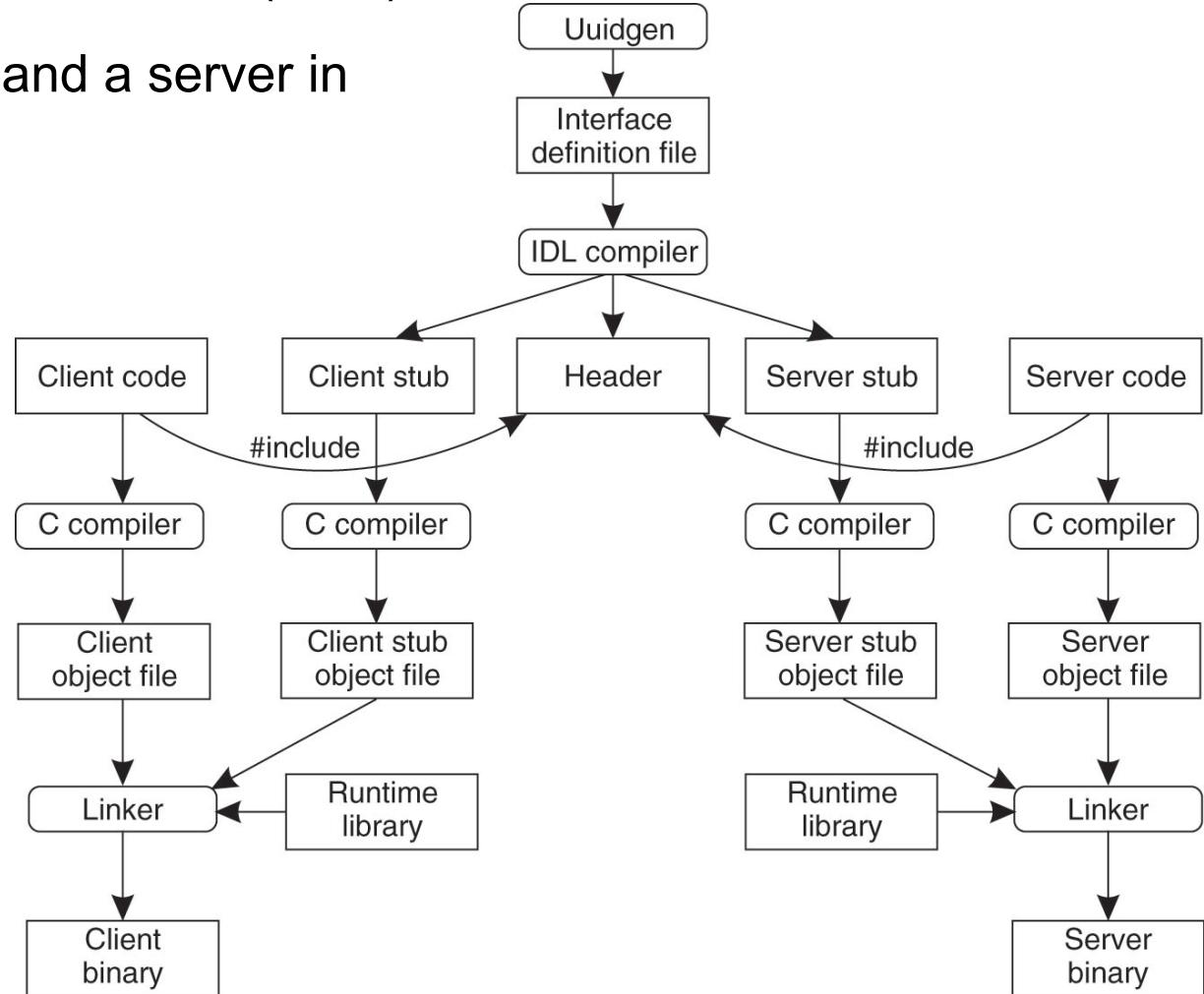
Asynchronous RPC

- › Interaction through two asynchronous RPCs: with deferred returned result



Distributed Computing Environment (DCE): middleware for UNIX/Windows...

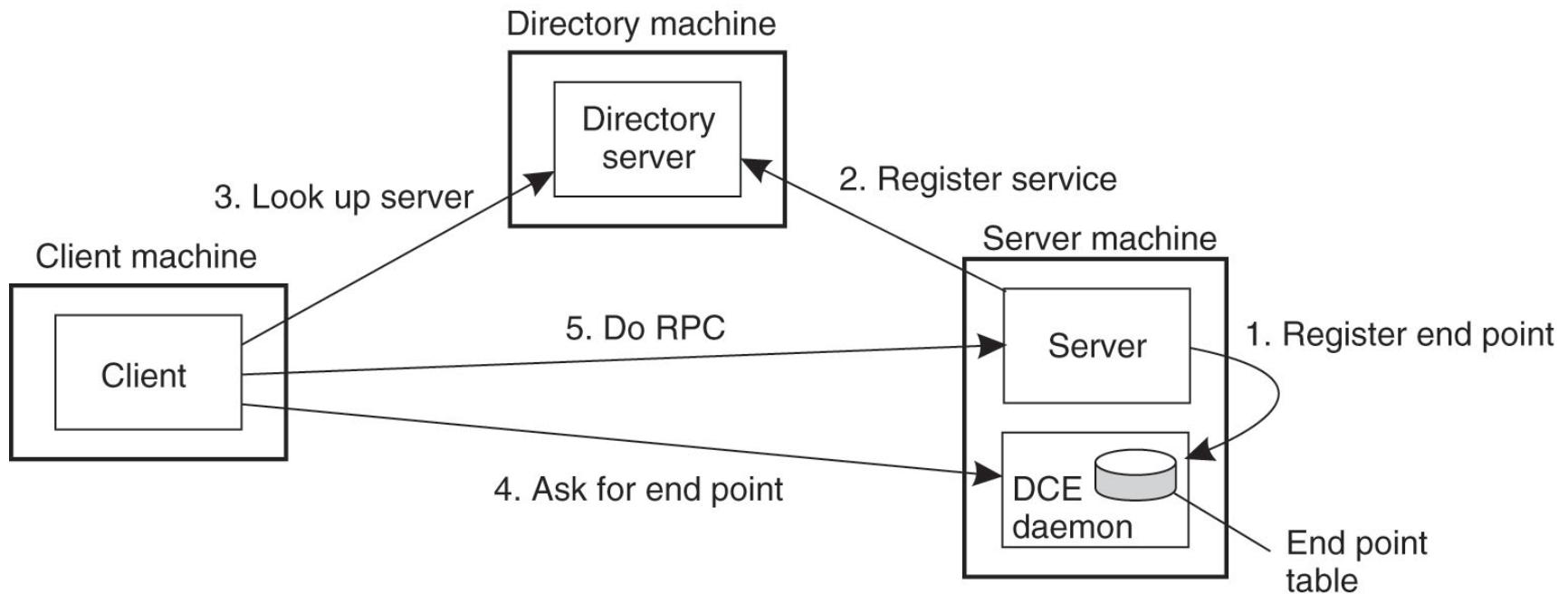
- › Steps in writing a client and a server in DCE/RPC



Distributed Computing Environment (DCE)

› Client and server binding:

- Locate server: using a directory machine indicating which server runs the service
- Locate process on server: the DCE daemon maintains a (server, port) table



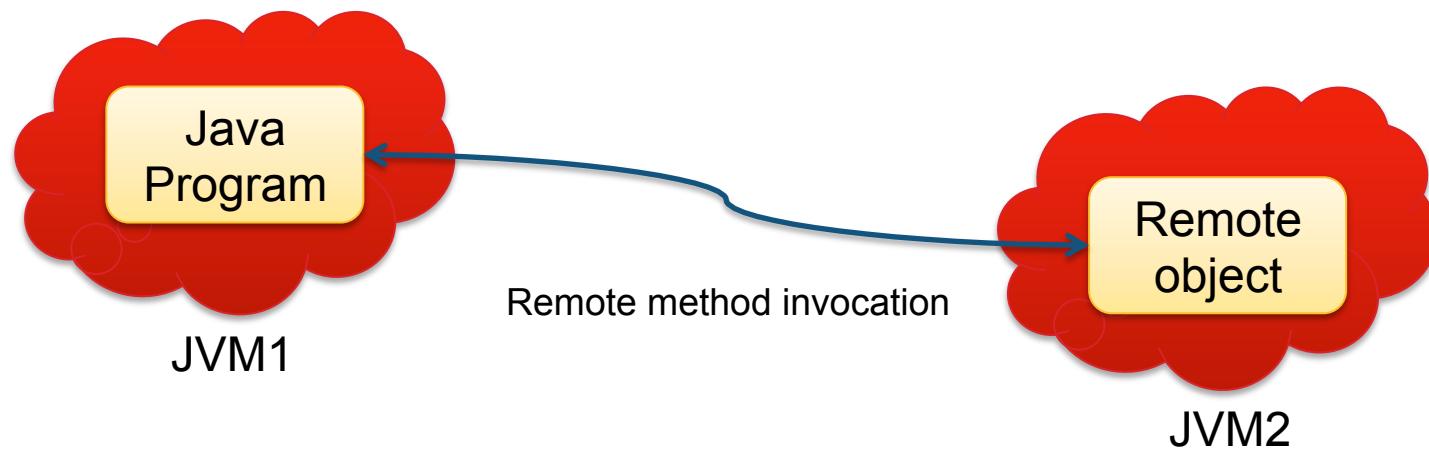


THE UNIVERSITY OF
SYDNEY

Remote Method Invocation (RMI)

RMI: A Java feature similar to RPC

- › An RMI allows a thread to invoke a method on a remote object
- › A *remote object* is an object residing in a different Java Virtual Machine (JVM)



Two main differences between RMI and RPC

1. Procedure vs. method

- RPC is **procedural**: only functions or procedures may be called
- RMI is **object-based**: invocation of methods on remote objects

2. Data structures vs. primitive data types

- RPC parameters are generally **data structures**
- RMI parameters can be **primitive data type** (e.g., int, boolean) as well as objects

Two main differences between RMI and RPC

- › *Stub*: a proxy for the remote object that resides at the client
- › *Skeleton*: responsible for unmarshalling the message on the server-side
- › *Parcel*: the name of the method and the marshalled parameters



1. The client-side stub **creates** the parcel, sends it to the server
2. The skeleton **unmarshals** the parameters and invokes the desired method on the server
3. The skeleton **marshals** the return value (or the exception, if any) into a parcel and returns this parcel
4. The stub **unmarshals** the return value and passes it to the client

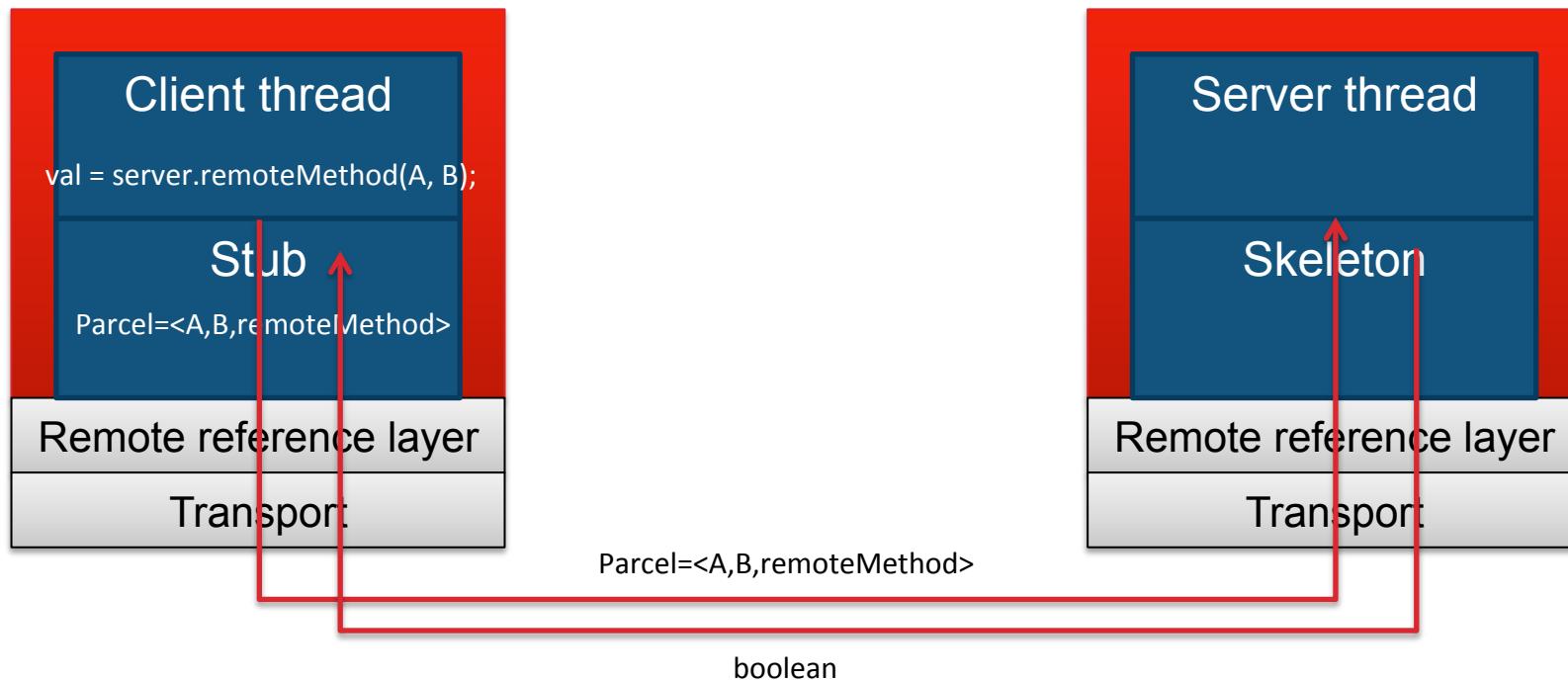
Passing parameters

- › Call-by-copy/restore:
 - Local (or non-remote) objects are passed by copy using object serialization
 - The local object must implement the `java.io.Serializable` interface (most of them do)
- › Call-by-reference:
 - Remote objects are passed by reference
 - It allows the receiver to alter the state of the remote object as well as invokes its remote methods

Example: a client invokes `remoteMethod(Object, Object)` returning a boolean

- › Client executes the following statement:

```
boolean val = server.remoteMethod(A, B);
```



Example: a date server program using RMI

- › The RemoteDate interface: specifies the methods that can be invoked remotely.

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

public interface RemoteDate extends Remote {
    public Date getDate() throws RemoteException;
}
```

- › In our example, only the method getDate() can be called remotely
- › All methods declared in the interface must throw the exception java.rmi.RemoteException
- › The class that defines the remote object must implement the remote interface

Example: a date server program using RMI (con't)

- › Implementation of the RemoteDate interface: the class must extend the UnicastRemoteObject for listening to request using RMI default sockets

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class RemoteDateImpl extends UnicastRemoteObject implements remoteDate {
    public RemoteDateImpl() throws RemoteException {}
    public Date getDate() throws RemoteException {
        return new Date();
    }

    public static void main(String args[]) {
        try {
            RemoteDate dateServer = new RemoteDateImpl();
            // Bind this object instance to the name "RMIDateObject"
            Naming.rebind("RMIDateObject", dateServer);
        } catch (Exception e) { System.err.println(e); }
    }
}
```

Example: a date server program using RMI (con't)

- › The main method creates an instance of the object and registers with the RMI registry running on the server via rebind() method

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class RemoteDateImpl extends UnicastRemoteObject implements remoteDate {
    public RemoteDateImpl() throws RemoteException {}
    public Date getDate() throws RemoteException {
        return new Date();
    }

    public static void main(String args[]) {
        try {
            RemoteDate dateServer = new RemoteDateImpl();
            // Bind this object instance to the name "RMIDateObject"
            Naming.rebind("RMIDateObject", dateServer);
        } catch (Exception e) { System.err.println(e); }
    }
}
```

Example: a date server program using RMI (con't)

- › A default constructor must be created, it throws a RemoteException if the communication problem prevents the remote object from being exported

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class RemoteDateImpl extends UnicastRemoteObject implements remoteDate {
    public Date RemoteDateImpl() throws RemoteException {}
    public Date getDate() throws RemoteException {
        return new Date();
    }

    public static void main(String args[]) {
        try {
            RemoteDate dateServer = new RemoteDateImpl();
            // Bind this object instance to the name "RMIDateObject"
            Naming.rebind("RMIDateObject", dateServer);
        } catch (Exception e) { System.err.println(e); }
    }
}
```

Example: a date server program using RMI (con't)

- › The RMI client: gets a proxy reference to the (rebound) object from the RMI registry running on the server with a URL lookup scheme

```
import java.rmi.*;  
  
public class RMIClient {  
    static final String server = "127.0.0.1";  
  
    public static void main(String args[]) {  
        try {  
            String host = "rmi://" + server + "/RMIDateObject";  
  
            RemoteDate dateServer = (RemoteDate)Naming.lookup(host);  
            System.out.println(dateServer.getDate());  
        }  
        catch (Exception e) {  
            System.err.println(e);  
        }  
    }  
}
```

Example: a date server program using RMI (con't)

- › The RMI client: once it has the proxy reference of the object, it invokes the remote method getDate(); lookup and remote method may throw exception

```
import java.rmi.*;  
  
public class RMIClient {  
    static final String server = "127.0.0.1";  
  
    public static void main(String args[]) {  
        try {  
            String host = "rmi://" + server + "/RMIDateObject";  
  
            RemoteDate dateServer = (RemoteDate)Naming.lookup(host);  
            System.out.println(dateServer.getDate());  
        }  
        catch (Exception e) {  
            System.err.println(e);  
        }  
    }  
}
```

Example: a date server program using RMI (con't)

- › Executing on the same machine:
 - The registry should be started and run in the background using
 - rmiregistry & (on unix-like machines)
 - start rmiregistry (on windows)
- › An instance of the remote object has now to be created using
 - java RemoteDateImpl
- › Start the client that will get a proxy reference to the remote object
 - java RMIClient

```
> rmiregistry &
> java RemoteDateImpl
> java RMIClient
```



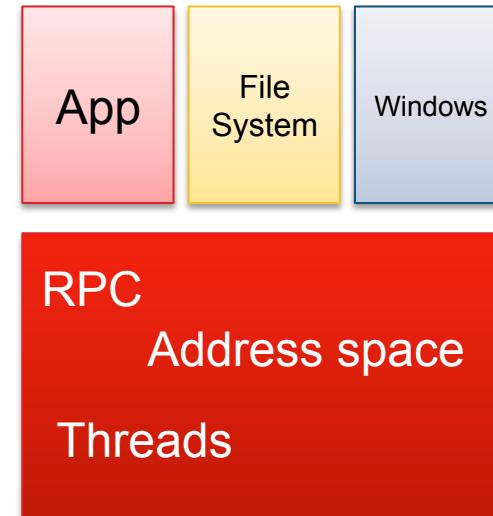
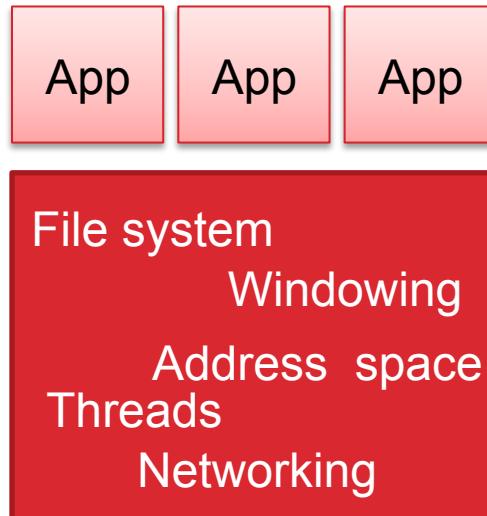
THE UNIVERSITY OF
SYDNEY

RPC-like Techniques

RPC-like techniques vs. Sockets

- › **Sockets** must manage the socket connection, including **opening** and **closing** the socket and **establishing** an **InputStream** and **OutputStream** to **read** from and **write** to the socket
- › The design of the client using **RMI** is much **simpler**, it must simply get a proxy for the remote object, to invoke the remote method as it would invoke an ordinary local method
- › Other RPC-like techniques (besides RPC and RMI)
 - CORBA (Common Object Request Broker Architecture)
 - DCOM (Distributed COM)

Application: Microkernel operating system



› Monolithic structure

- 2 Levels: lot of stuff running in kernel mode (Windowing in kernel in Windows for perf reason)
- Applications running in user mode
- One faulty component crashes the system

› Microkernel structure

- Thread package, address space, RPC mechanisms run in kernel mode
- File system runs at user level
- Bugs are isolated
- Clean API: components can be migrated



- › Message losses can have dramatic consequences
- › TCP/IP protocol suite hides these losses from the application level
- › Socket, RPC, RMI use TCP/IP
 - Sockets are complex
 - RPC and RMI are more transparent for the client