Lecture 4: RPC

Remote Procedure Call

Coulouris et al: Chapter 5

(but see Chapter 4 for “marshalling”)

TVS: Section 4.2 (see 10.3.4 for RMI)

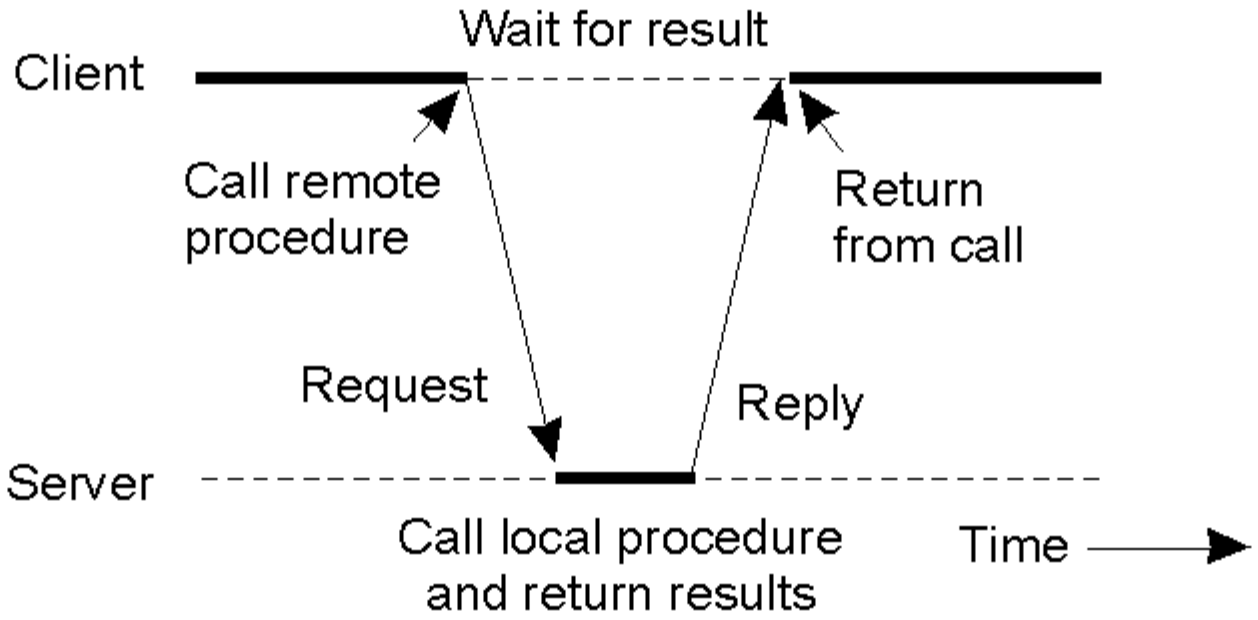
Basics

- Distributed systems can be built using messages with *send* and *receive*
- But these are rather low-level constructs
- Remote Procedure Call (1984 – Birrell & Nelson) provides a higher-level alternative
- It provides *access transparency*, i.e. the use of a local service (e.g. provided by the OS) has the same form as use of a non-local one.

Basics (continued)

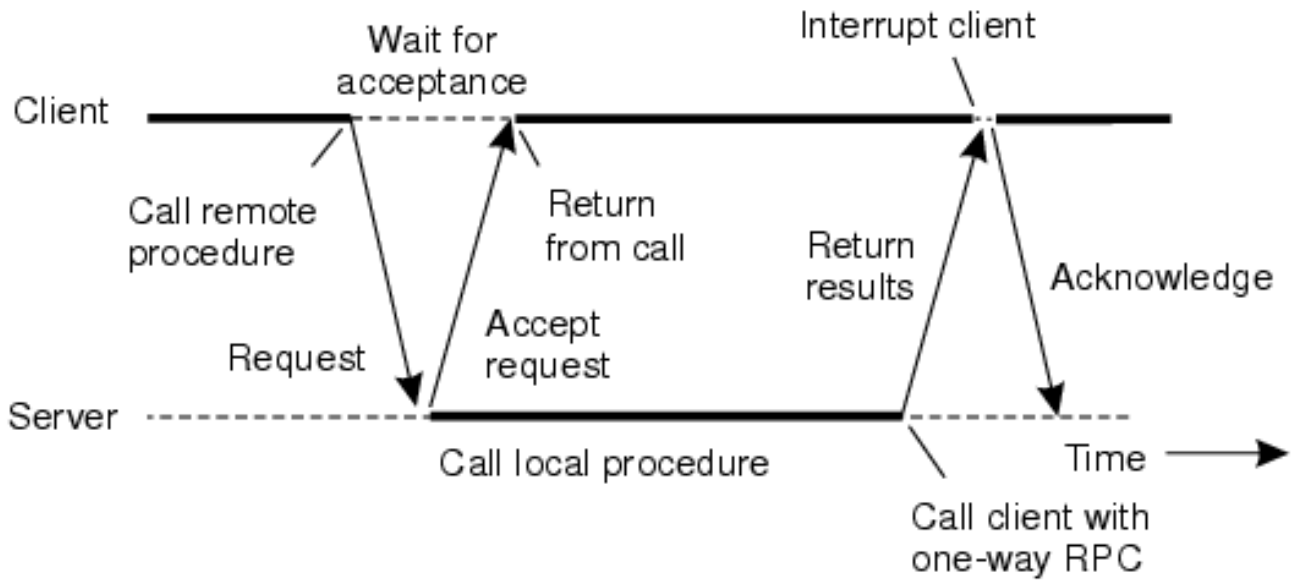
- So a process on the client “calls” a process on the server to execute the code of the procedure (providing some service)
- Arguments are sent in the message to the server
- The result is sent in the reply from the server to the client

RPC between Client and Server (synchronous)



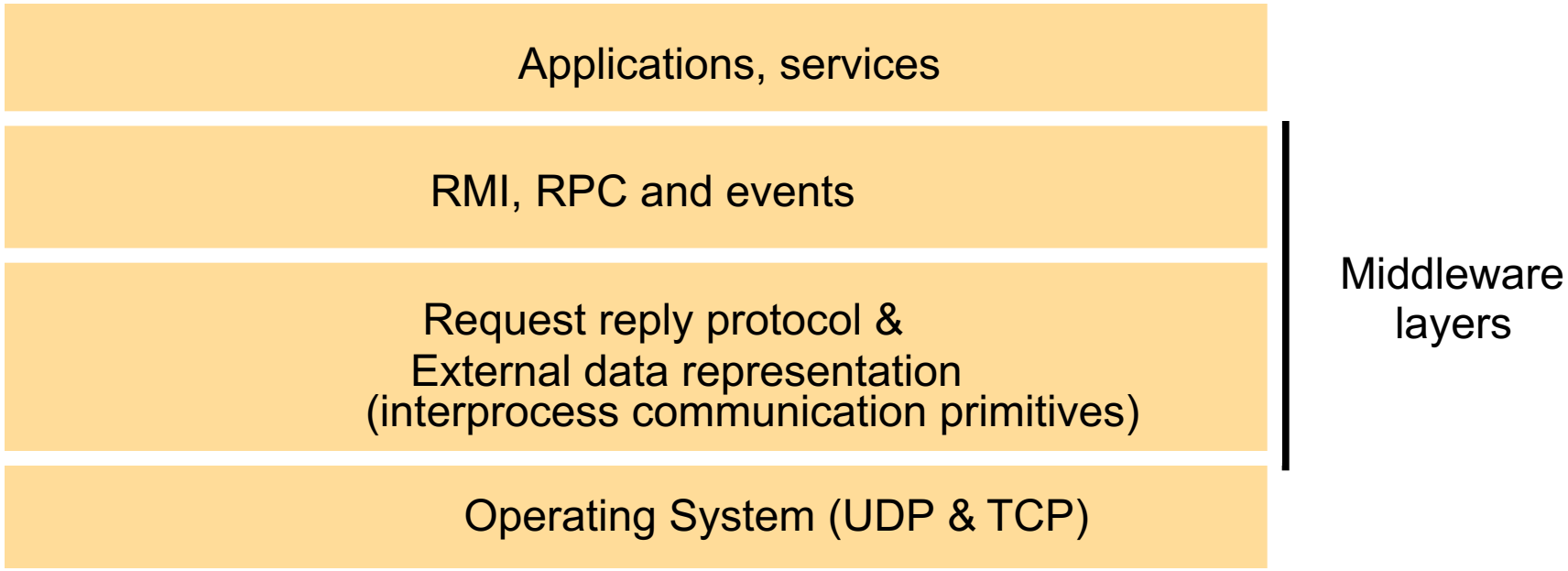
TVS Fig 4-6 (and 4-10a)

RPC between Client and Server (asynchronous)



TVS Fig 4-11

CDK Figure 5.1
Middleware layers



Details: arguments

- Client and server in different computers => cannot use addresses!
- Value parameters are OK
- Caller and callee need to agree on the format of messages!

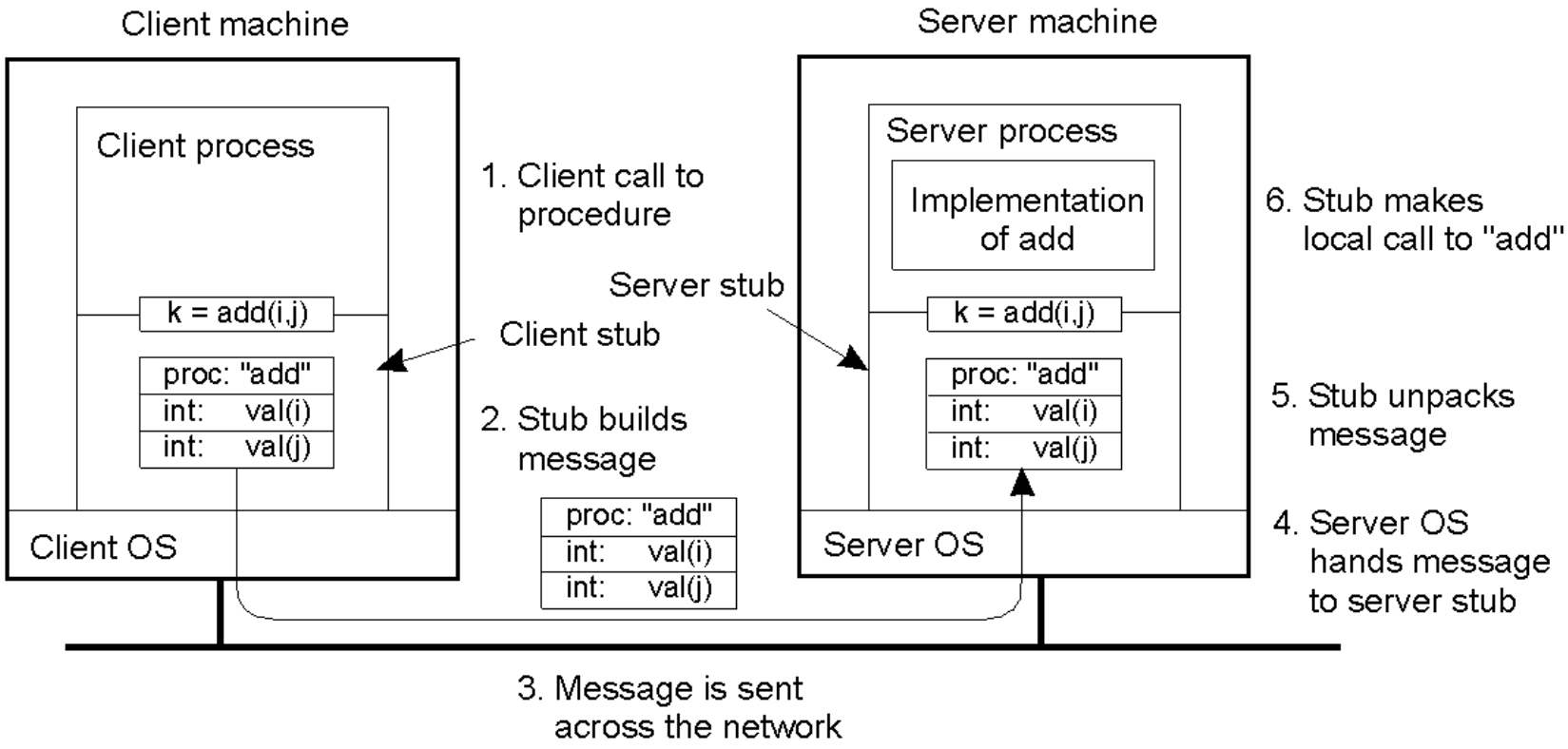
Stubs

- Instead of the code to provide the service, the client machine has a stub procedure. That:
  - puts the arguments in a message
  - sends that message to the server
  - waits for the reply message
  - unpacks the result
  - returns to the application call

Server stub (or skeleton)

- Transparency works both ways!
- Code in the server to provide the service should also be "normal" – so it can be used by other code on the server to provide the same service locally ….
- So the RPC message goes to a server stub – code on the server

First Six Steps of RPC
TVS: Figure 4-7



Remaining Steps

- 7: **add** executes and returns answer to server stub
- 8: server stub composes reply message containing this answer
- 9: message is sent across the network
- 10: client OS passes reply to client stub
- 11: client stub unpacks and returns result

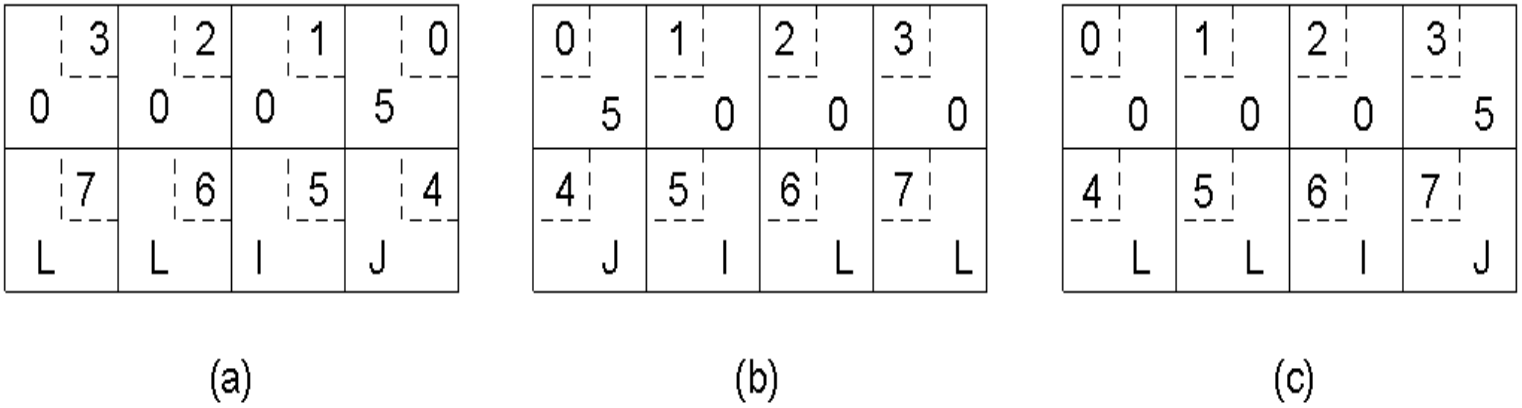
Parameter Marshalling

- Packing parameters into a message is known as *parameter marshalling*
- The inverse operation is *unmarshalling*
- These operations need to take into account problems associated with having different machines and different languages in use in the network

Representational issues

- Client and server can use different ways to represent values:
  - Floating point formats
  - Big-endian (Western style) vs little-endian (e.g., Intel)
  - Character sets
  - Size issues (e.g. 64 bit ints or 32 bit ints)

Passing Value Parameters
TVS Figure 4-8



a) Original message on the Pentium ([12]=1+2x256)

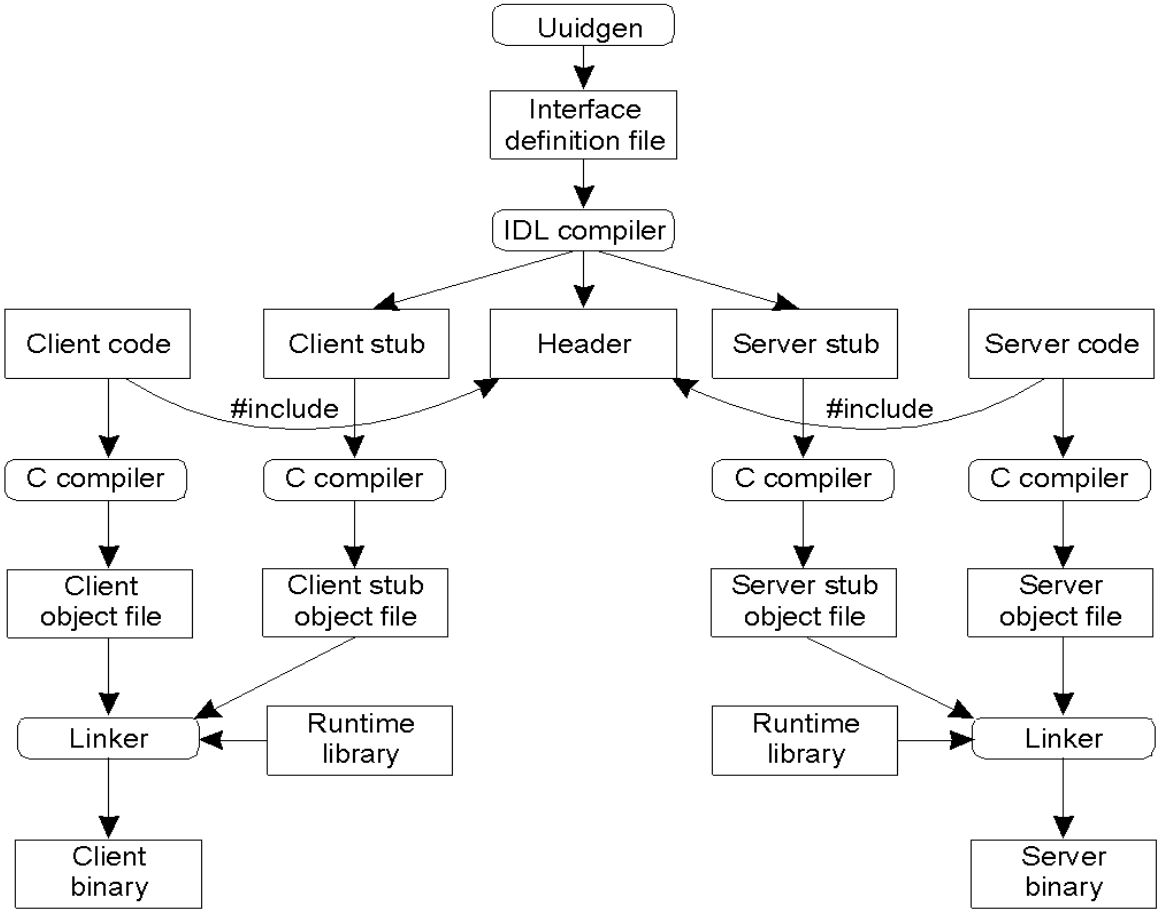
b) The message after receipt on the SPARC ([12] = 1x256+2)

c) The message after being inverted. The little numbers in boxes indicate the address of each byte

Generating Stubs

- The code for the stubs can be generated once the specification of the procedure is known.
- This should ideally be done in a language independent way – so that the remote service is available to many languages
- Interface Definition Language (IDL)

Producing a Client and a Server
TVS Figure 4-12



Uuidgen?

- The purpose of this is to generate a unique identifier which is
  - Put in the header file
  - Can be used to check that the client and server stubs are compatible, i.e. to protect against errors caused by updating one and not the other

Features of an IDL

- There are a number of IDLs
  - CORBA (Common Object Request Broker Architecture)
  - Sun (noted for NFS)
  - DCE (Distributed Computing Environment)
- Syntax of interface – but commonly with extra information (e.g. *in*, *out*, *inout*) about reference parameters

```
// Example – CORBA IDL
// In file Person.idl
struct Person {
    string name;
    string place;
    long year;
} ;
interface PersonList {
    readonly attribute string listname;
    void addPerson(in Person p) ;
    void getPerson(in string name,
                   out Person p);
    long number();
};
```

Remote Method Invocation (RMI)

- Java includes RMI in its API
- Marshalling is simpler: Java <=> Java
- It requires that objects sent as arguments or results be *serializable,* i.e. there is a standard external form for them. This allows arbitrarily complex structures (not possible with copy/restore parameters)

Remote Object References

- A major difference between RMI and RPC is that there are, in RMI, references to remote objects!
- In the lab, **`RemoteServer`** is an interface used to declare **`stub`** which is such a thing
- The interface lists the methods which can be invoked on it
- What are Remote Object References?
  - (CDK 4.3.4) They must be
    - system-wide
    - not reused for other objects

RMI implementation

- As with RPC, the client has a stub (called here a *proxy)* for each remote class instance – to marshal arguments to, and unmarshal results from, method calls
- This communicates with the *dispatcher* for the class, which forwards to the *skeleton* which implements unmarshalling etc.

Dispatcher / Skeleton

- Server has a dispatcher & a skeleton for each class of remote object
- The dispatcher receives the incoming message, and uses its method info to pass it to the right method in the skeleton. (Dispatcher and proxy both work from the remote interface to get same method info!)
- Skeleton implements methods of the remote interface to unmarshall arguments and invoke the corresponding method in the servant. It then marshalls the result (or any exceptions thrown) into a reply message to the proxy.

rmiregistry

- This is how a server makes a remote object available to clients
- A string is bound to the remote object, and the clients interrogate the registry using the string
- The client must know the server machine name and the port the registry is on (there is a default)

Conclusion

- When processes on different computers communicate:
  - need to agree on the format of messages
  - need to take into account differences of machines
- Client and Server stubs are implemented
- Interfaces can be specified by means of an IDL
- Reading: Coulouris (ed. 4 or 5): Chapter 5 (but see Chapter 4 for "marshalling"); Tanenbaum: Section 4.2 (see 10.3.4 for RMI)