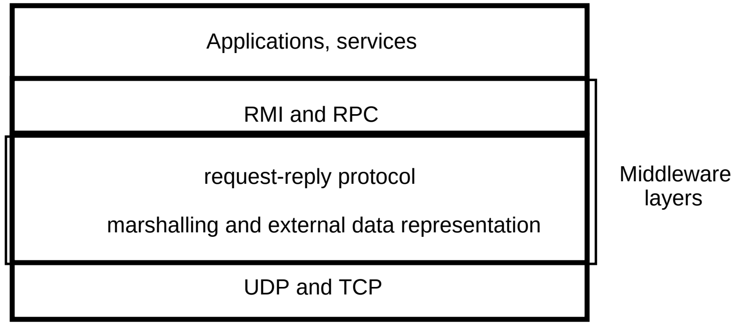
Distributed Computing

Interprocess Communication and Remote Procedure Invocation

Dr. Shen Zhiqi
zqshen@ntu.edu.sg

Middleware layers



Applications, services

RMI and RPC

request-reply protocol

marshalling and external data representation

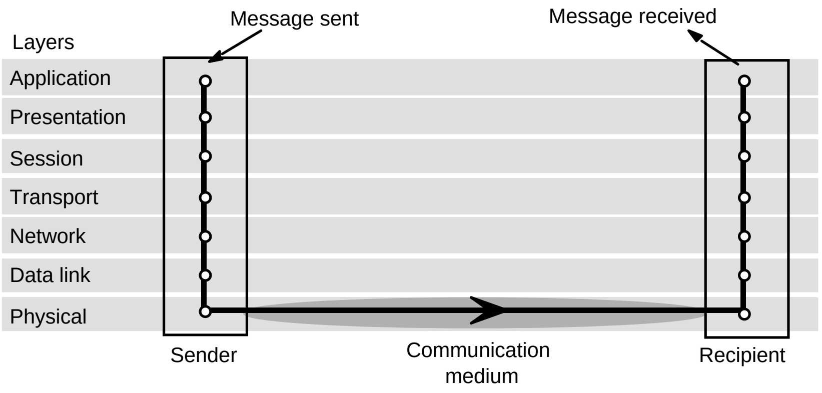
UDP and TCP

Middleware layers

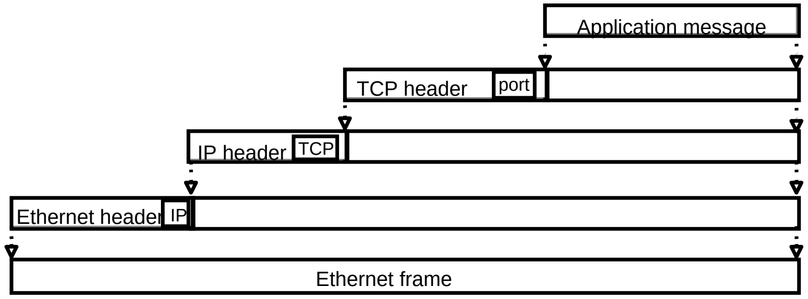
OSI protocol summary

Layer	*Description*	*Examples*
Application	Protocols that are designed to meet the communication requirements of specific applications, often defining the interface to a service.	HTTP,FTP, SMTP, CORBA IIOP
Presentation	Protocols at this level transmit data in a network representation that is independent of the representations used in individual computers, which may differ. Encryption is also performed in this layer, if required.	Secure Sockets (SSL),CORBA Data Rep.
Session	At this level reliability and adaptation are performed, such as detection of failures and automatic recovery.	
Transport	This is the lowest level at which messages (rather than packets) are handled. Messages are addressed to communication ports attached to processes, Protocols in this layer may be connection-oriented or connectionless.	TCP, UDP
Network	Transfers data packets between computers in a specific network. In a WAN or an internetwork this involves the generation of a route passing through routers. In a single LAN no routing is required.	IP, ATM virtual circuits
Data link	Responsible for transmission of packets between nodes that are directly connected by a physical link. In a WAN transmission is between pairs of routers or between routers and hosts. In a LAN it is between any pair of hosts.	Ethernet MAC, ATM cell transfer, PPP
Physical	The circuits and hardware that drive the network. It transmits sequences of binary data by analogue signalling, using amplitude or frequency modulation of electrical signals (on cable circuits), light signals (on fibre optic circuits) or other electromagnetic signals (on radio and microwave circuits).	Ethernet base- band signalling, ISDN

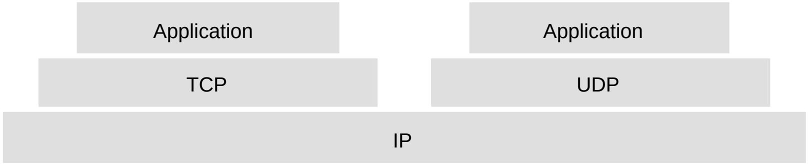
Protocol layers in the ISO Open Systems Interconnection (OSI) model



Encapsulation in a message transmitted via TCP over an Ethernet



The programmer's conceptual view of a TCP/IP Internet



IP

- IP → best-effort, unreliable, connectionless
  - Remembers nothing about a packet after it has sent it
  - Checksum computed on header only
- No assumptions about the underlying physical medium
  - Serial link, Ethernet, Token ring, X.25, ATM, wireless CDPD, …

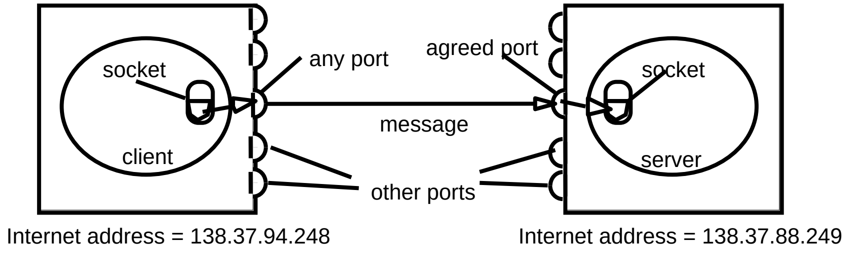
TCP and UDP

- TCP → reliable connection-oriented service
  - Segments are sent in IP datagrams
  - Checksum of data in each segment
  - Sequence # of the 1st byte in the segment
  - Acknowledge-and-retransmit mechanism
- UDP:
  - (optional) checksum

Interprocess Communication

- Synchronous and asynchronous communication
- Message destinations
  - Port
  - Name
- Reliability
- Ordering

Sockets and ports



socket

client

any port

agreed port

socket

server

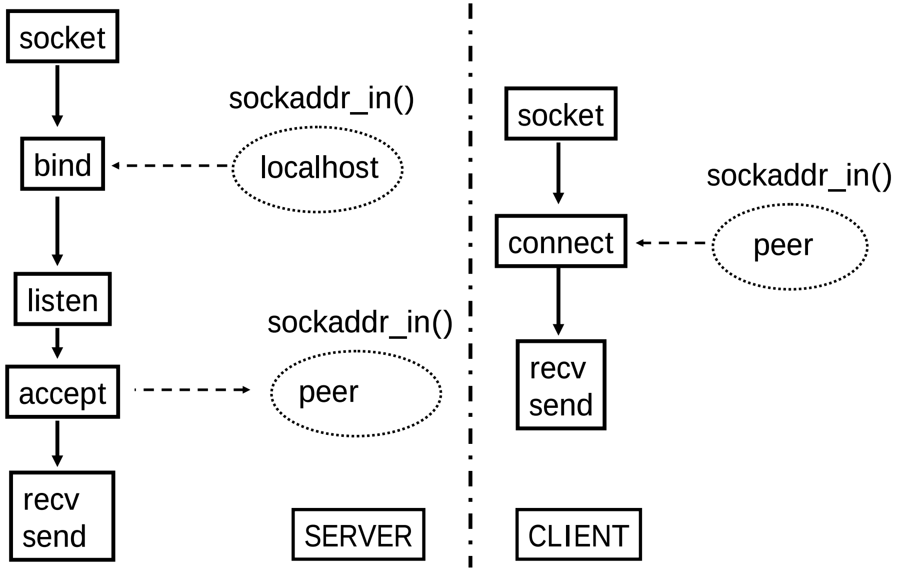
message

other ports

Internet address = 138.37.94.248

Internet address = 138.37.88.249

Basic socket calls



socket

bind

listen

accept

recv
send

sockaddr_in()

localhost

sockaddr_in()

peer

SERVER

socket

connect

recv
send

sockaddr_in()

peer

CLIENT

External Data Representation (I)

- External data representation: an agreed, platform-independent, standard for the representation of data structures and primitive values.
- Data structures:
  - “flattened” on transmission
  - rebuilt upon reception
- Primitive data types:
  - byte order
  - ASCII vs UNICODE (2 bytes per character)

Marshalling & Unmarshalling

- Marshalling: the act of taking a collection of data items (platform dependent) and assembling them into the external data representation (platform independent).
- Unmarshalling: the process of disassembling data that is in external data representation form, into a locally interpretable form.

External Data Representation (II)

- XDR (RFC 1832), CDR (CORBA), Java:
  - data -> byte stream
  - object references

IP address	port	time	object ID	interface ID

- HTTP/MIME:
  - data -> ASCII text

CORBA CDR
for constructed types

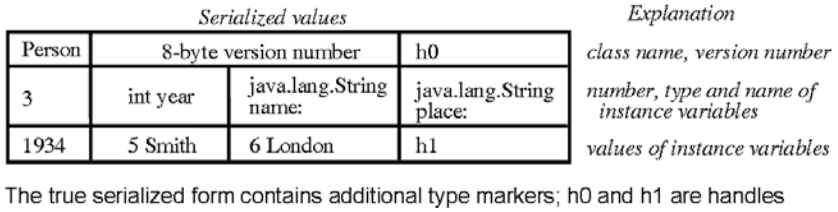
Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components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

CORBA CDR example:

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}

Indication of Java serialized form



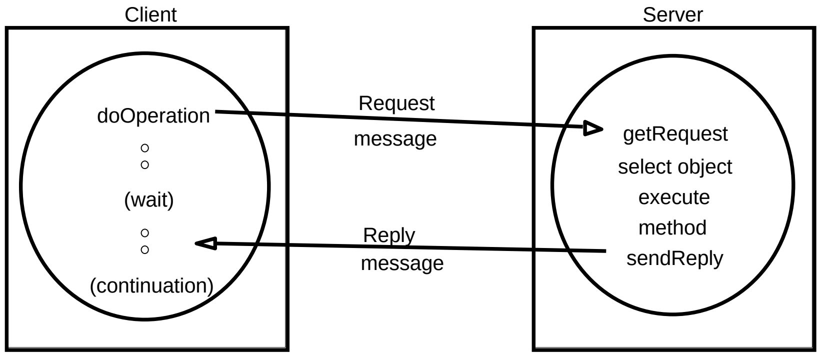
Serialized values				Explanation
Person	8-byte version number		h0	class name, version number
3	int year	java.lang.String name:	java.lang.String place:	number, type and name of instance variables
1934	5 Smith	6 London	h1	values of instance variables

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

XML definition of the Person structure

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

Client – Server Communication Request-Reply Protocol



Operations of 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.

Request-reply message structure

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

HTTP request message

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

HTTP reply message

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

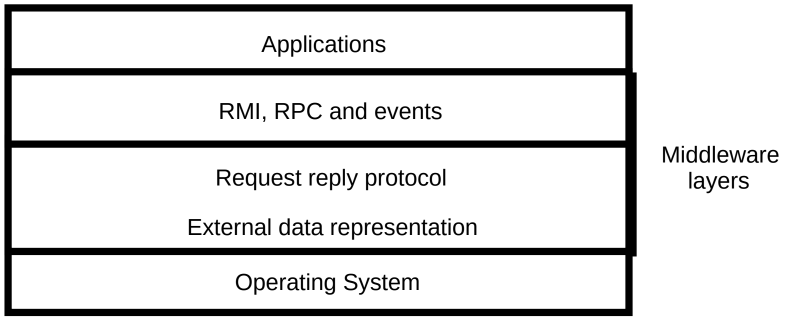
RPC exchange protocols

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

Group Communication

- Multicasting: 1-to-many comm. pattern
  - Applications:
    - replicated services (better fault tolerance)
    - discovery of services
    - replicated data (better performance)
    - propagation of event notifications
  - Failure model:
    - depends on implementation:
      - IP multicast (UDP datagrams): omission failures
        - class-D Inet addresses: “1110” bit prefix
        - TTL (time time to live)
      - reliable multicast
      - ordered multicast
        - FIFO
        - Causal
        - Total

Middleware layers



Interfaces

- Interface: provides a definition of the signatures of a set of methods (i.e., the types of their arguments, return values, and exceptions) without specifying their implementation.
  - Service Interfaces
  - Remote Interfaces

Interfaces

- Service Interfaces: refer to the specification of the procedures offered by a server, defining the types of the input and output arguments of each of the procedures.
- Remote Interfaces: specify the methods of an object available for remote invocation.
  - an interface definition language (or IDL) is used to specify remote interfaces. E.g. CORBA IDL.
  - Java RMI would have a class, but CORBA has a *struct*

CORBA IDL example

```
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();
};
```

Local Objects

- Object
  - consists of a set of data and a set of methods.
  - E.g., C++ object and Java object
- Object reference
  - an identifier via which objects can be accessed.

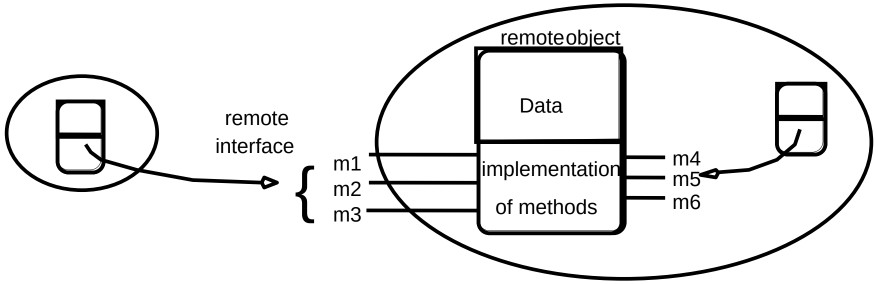
Remote Objects

- Remote objects
  - objects that can receive remote invocations.
- Remote object reference
  - an identifier that can be used globally *throughout a distributed system* to refer to a particular unique remote object.
- Remote interface
  - Every remote object has a remote interface that specifies which of its methods can be invoked remotely.

Remote method invocation

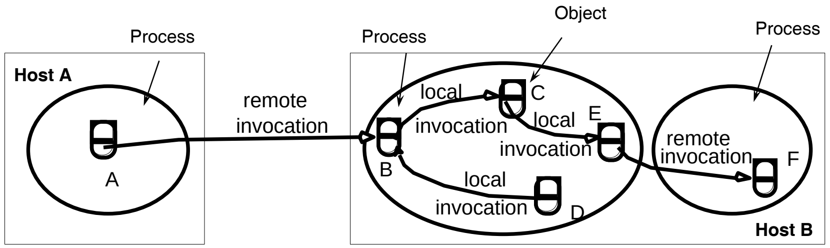
- Remote method invocation
  - Method invocations between objects *in different processes (processes may be on the same or different hosts)*.
  - Remote Procedure Call (RPC), which is between different processes (may be on same or different hosts)

A Remote Object and Its Remote Interface



Example Remote Object reference=(IP,port,objectnumber,signature,time)

Remote and Local Method Invocations



Local invocation=between objects on same process.
Has *exactly once* semantics

Remote invocation=between objects on different processes.
Ideally also want *exactly once* semantics for remote invocations

Each process contains objects, some of which can receive remote invocations, others only local invocations

Invocation semantics

- Semantics:
  - Zero-or-More ("Maybe")
  - At-Least-Once
    - Keep requesting RPC until valid response arrives at client
  - At-Most-Once
    - No reply may mean that no execution took place
  - Exactly-Once
    - Idempotent vs non-idempotent procedures
    - Require transaction processing techniques

Invocation Semantics

Whether or not to retransmit the request message until either a reply is received or the server is assumed to be failed

when retransmissions are used, whether to filter out duplicate requests at the server.

whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations

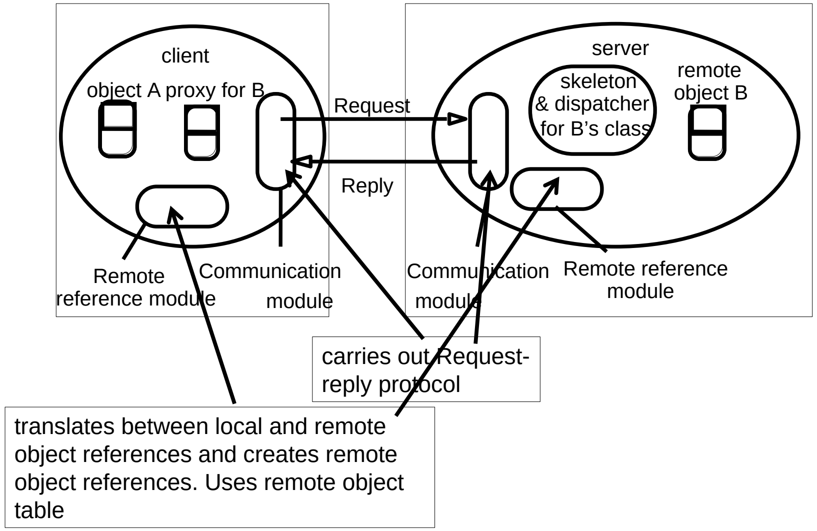
Fault tolera measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	(ok for *idempotent* operations) Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Idempotent=same result if applied repeatedly

Invocation semantics: failure model

- *Maybe*, *At-least-once* and *At-most-once* can suffer from crash failures when the server containing the remote object fails.
- *Maybe* - if no reply, the client does not know if method was executed or not
  - omission failures if the invocation or result message is lost
- *At-least-once* - the client gets a result (and the method was executed at least once) or an exception (no result)
  - arbitrary failures. If the invocation message is retransmitted, the remote object may execute the method more than once, possibly causing wrong values to be stored or returned.
  - if *idempotent* operations are used, arbitrary failures will not occur
- *At-most-once* - the client gets a result (and the method was executed exactly once) or an exception (instead of a result, in which case, the method was executed once or not at all)

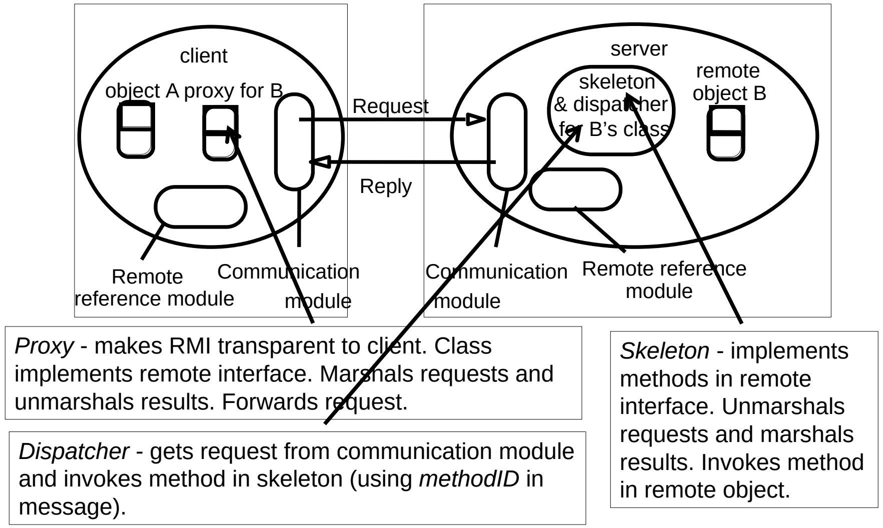
The architecture of remote method invocation



carries out Request-reply protocol

translates between local and remote object references and creates remote object references. Uses remote object table

The architecture of remote method invocation



Proxy - makes RMI transparent to client. Class implements remote interface. Marshals requests and unmarshals results. Forwards request.

Dispatcher - gets request from communication module and invokes method in skeleton (using *methodID* in message).

Skeleton - implements methods in remote interface. Unmarshals requests and marshals results. Invokes method in remote object.

Remote Reference Module

- Is responsible for translating between local and remote object references and for creating remote object references.
- Has a *remote object table*
  - An entry for each remote object held by any process. E.g., B at P2.
  - An entry for each local proxy. E.g., proxy-B at P1.
- When a new remote object is seen by the remote reference module, it creates a remote object reference and adds it to the table.
- When a remote object reference arrives in a request or reply message, the remote reference module is asked for the corresponding local object reference, which may refer to either a proxy or to a remote object.
- In case the remote object reference is not in the table, the RMI software creates a new proxy and asks the remote reference module to add it to the table.

Remote object reference

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

- a remote object reference must be unique in the distributed system and over time. It should not be reused after the object is deleted.
- the first two fields locate the object.
- the fourth field identifies the object within the process. its interface tells the receiver what methods it has (e.g. class *Method*)
- a remote object reference is created by a remote reference module when a reference is passed as argument or result to another process
  - – it will be stored in the corresponding proxy
  - – it will be passed in request messages to identify the remote object whose method is to be invoked

Proxy

- Is responsible of making RMI transparent to clients by behaving like a local object to the invoker.
  - – The proxy *implements* (Java term, not literally) the methods in the interface of the remote object that it represents. But,…
- Instead of executing an invocation, the proxy forwards it to a remote object.
  - – Each method of the proxy *marshals* the following into a request message: (i) a reference to the target object, (ii) its own method id and (iii) the argument values. Request message is sent to the target, then proxy awaits the reply message, *un-marshals* it and returns the results to the invoker.

Dispatcher and Skeleton

- Each process has one dispatcher, and a skeleton for each local object (actually, for the class).
- The dispatcher receives all request messages from the communication module.
  - – For the request message, it uses the method id to select the appropriate method in the appropriate skeleton, passing on the request message.
- Skeleton “implements” the methods in the remote interface.
  - – A skeleton method un-marshals the arguments in the request message and invokes the corresponding method in the remote object (the actual object).
  - – It waits for the invocation to complete and marshals the result, together with any exceptions, in a reply message.

Generation of Proxies, Dispatchers and Skeletons

- In CORBA, programmer specifies interfaces of remote objects in CORBA IDL; then, the interface compiler automatically generates code for proxies, dispatchers and skeletons.
- In Java RMI
  - – The programmer defines the set of methods offered by a remote object as a Java interface implemented in the remote object.
  - – The Java RMI compiler generates the proxy, dispatcher and skeleton classes from the class of the remote object.

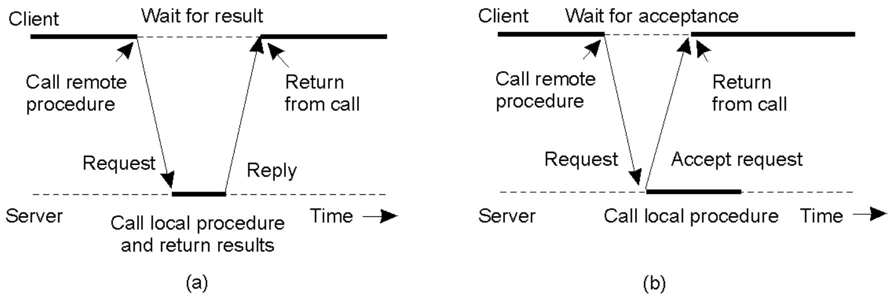
Steps of a Remote Method Invocation

1. Client makes a call to remote method
2. Proxy marshalls the request message
3. Client's OS sends message
4. Message is transmitted to server
5. Server receives the message, calls skeleton
6. Skeleton unmarshalls the message
7. Server processes the request
8. Skeleton marshalls the message
9. Server's OS sends message
10. Message is transmitted to client
11. Client receives the message
12. Proxy unmarshalls the message

Remote Procedure Call (RPC)

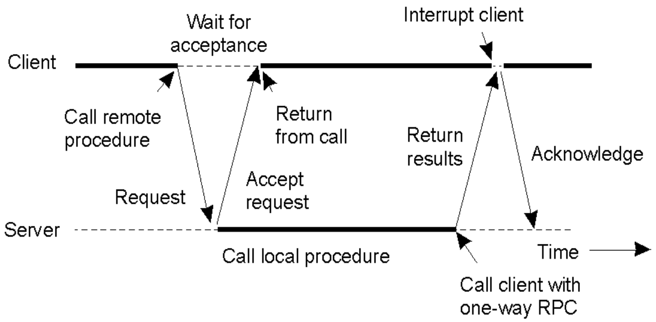
- Uniform, reusable, user-friendly, and action based.
  - Provide a familiar interface for the application developer
  - Implements the request-reply primitive
  - Format of the message is standard
  - Supports code reuse
- Client process calls for invocation of a procedure at the server process.
  - Semantics are similar to RMIs – at least once, at most once, maybe
  - Standard interface, independent of applications
- A library of reusable procedures, distributed over all sites.

Asynchronous RPC (I)



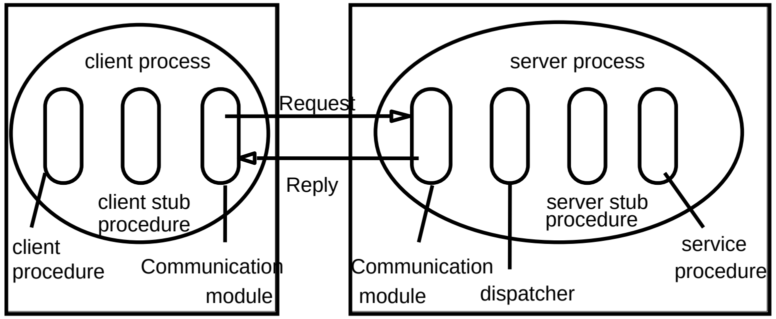
a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC

Asynchronous RPC (II)



- A client & server interacting through 2 asynchronous RPCs

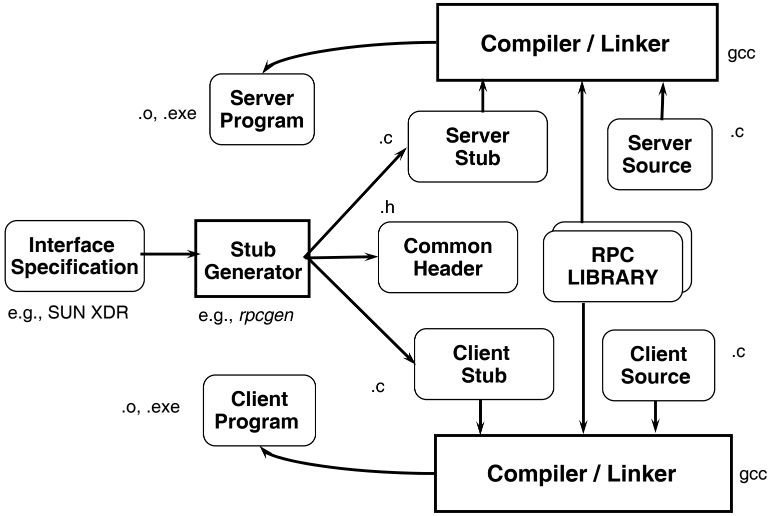
Client and Server Stub Procedures in RPC



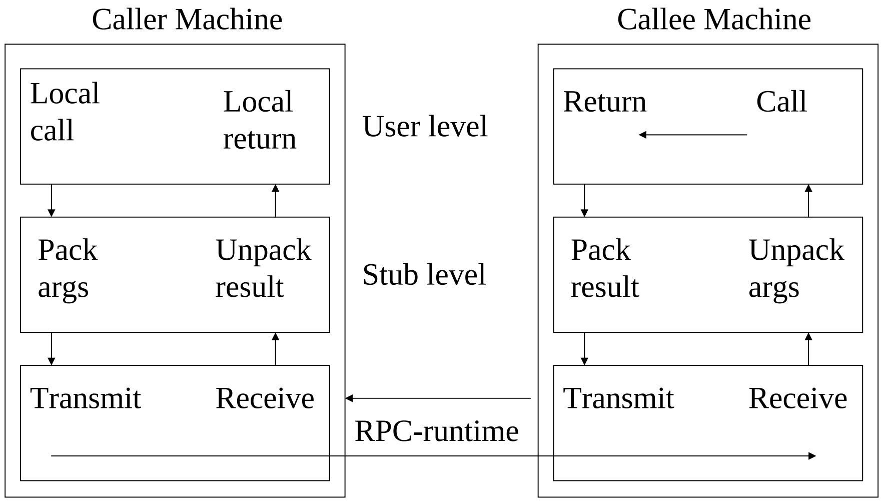
Stubs

- Stubs are generated automatically from interface specifications.
- Stubs hide details of (un)marshalling from application programmer & library code developer.
- Client Stubs perform marshalling into request messages and unmarshalling from reply messages
- Server Stubs perform unmarshalling from request messages and marshalling into reply messages
- Stubs also take care of communication & invocation

The Stub Generation Process



RPC Implementation



Steps of a Remote Procedure Call

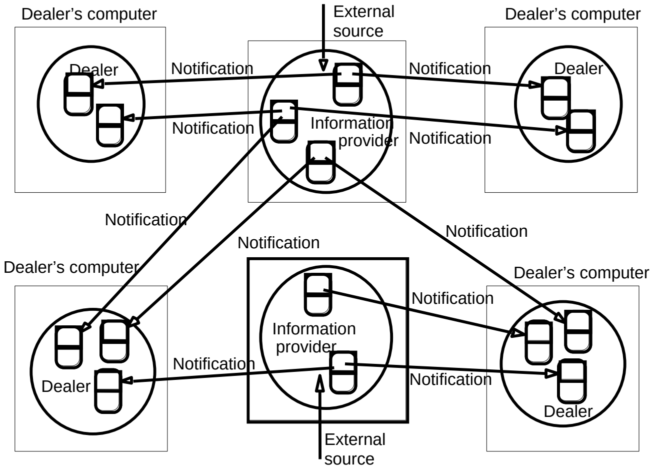
1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Events and Notifications

- Publish-subscribe paradigm
- Distributed event-based systems
  - Heterogeneous
  - Asynchronous
- Events
  - Types
  - Attributes

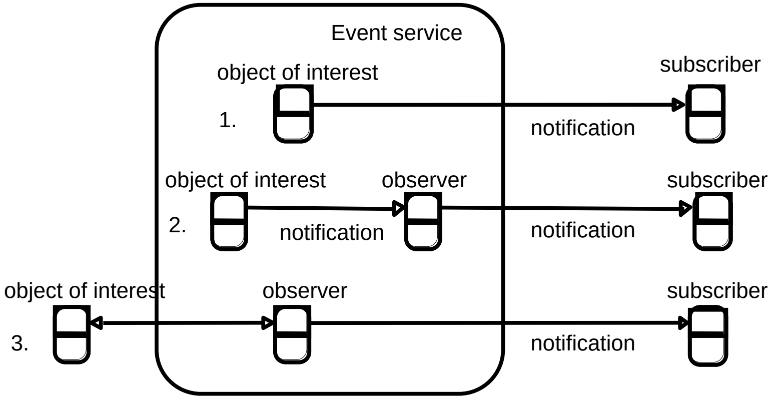
Dealing Room System

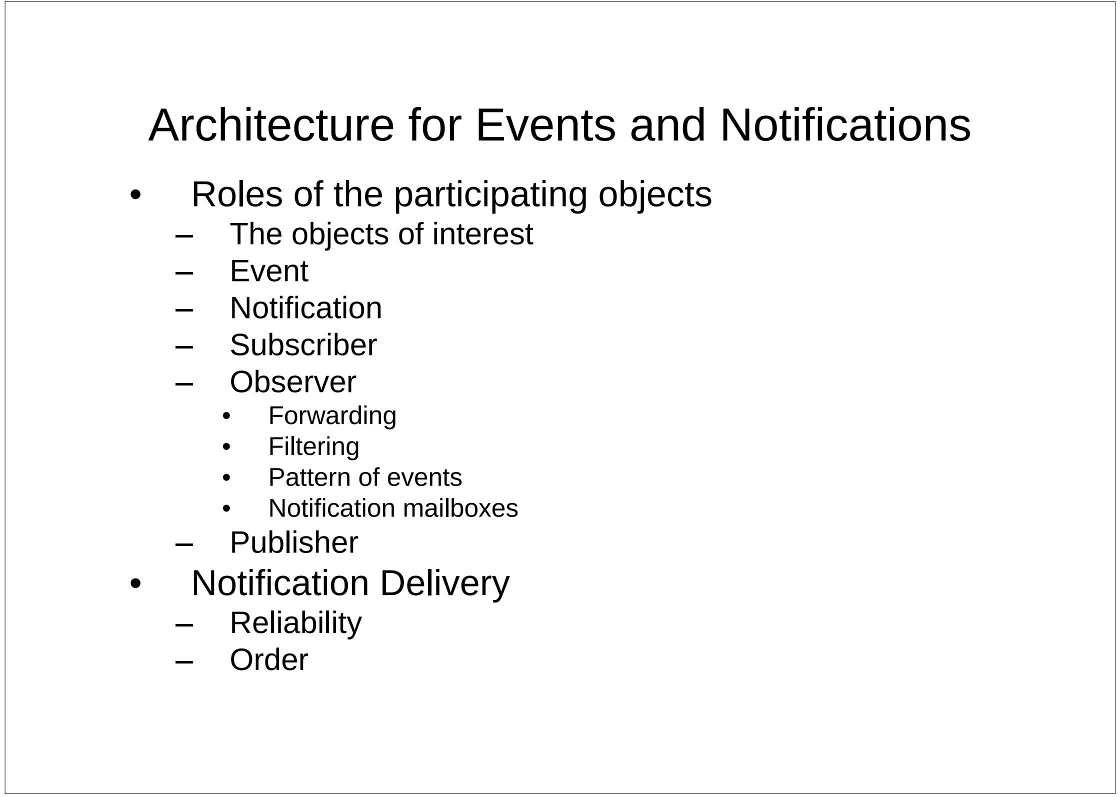
[Publish-Subscribe System]
e.g, stock market



At each dealer:
One object per
named stock

Architecture for Distributed Event Notification





Architecture for Events and Notifications

- Roles of the participating objects
  - The objects of interest
  - Event
  - Notification
  - Subscriber
  - Observer
    - Forwarding
    - Filtering
    - Pattern of events
    - Notification mailboxes
  - Publisher
- Notification Delivery
  - Reliability
  - Order