Interaction Styles: Goals of this lecture

- **Students have an overview of common interaction styles (communication models) and their properties**
- **Student have a basic knowledge how these styles are realized as middleware services and by middleware frameworks**
- **Students have seen examples of such styles**

Agenda

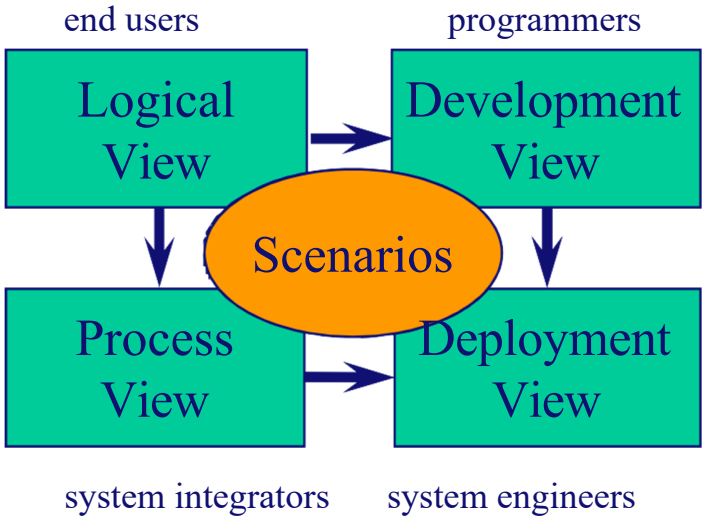
- Introduction
- Layered protocols and middleware
- Remote Invocation
  - (Remote) Procedure calls
  - (Remote) Method invocation
- Message Oriented Middleware (MOM)
- Streaming

Interaction styles / communication models

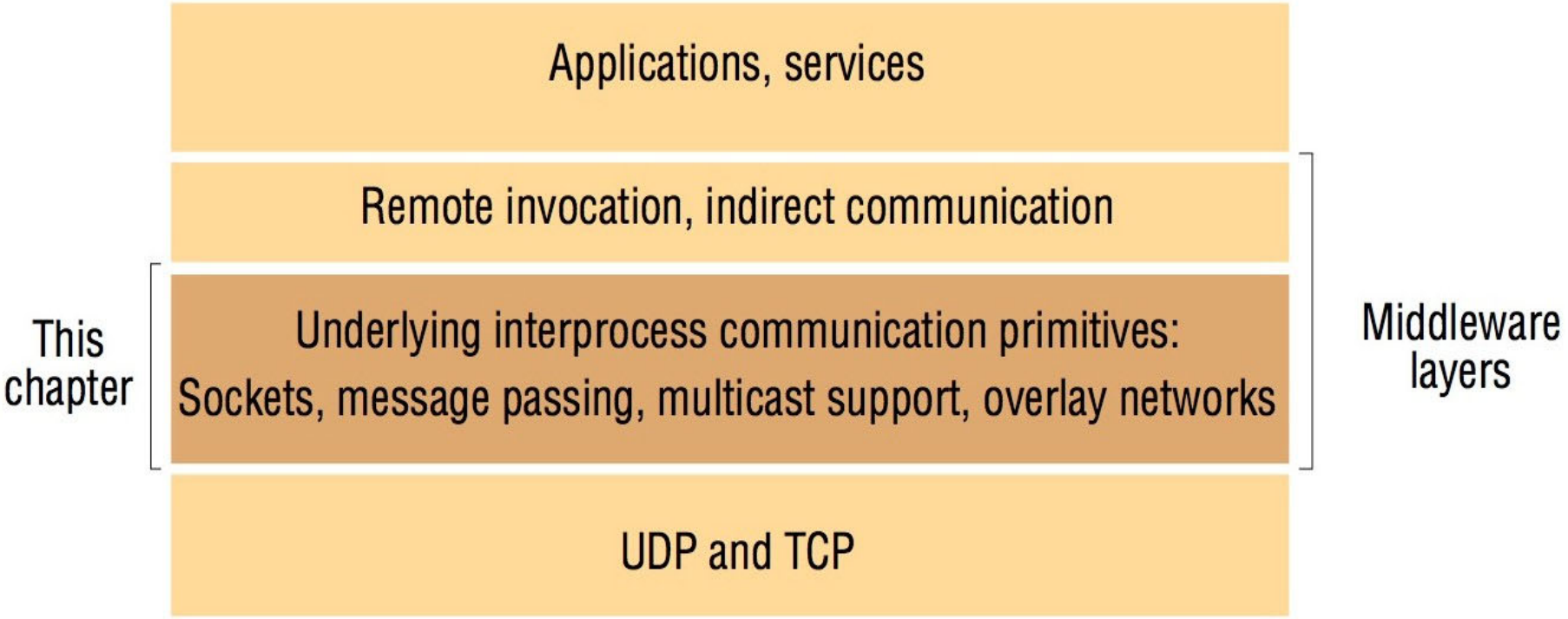
- Basic inter-process communication (IPC)
  - message passing, sockets
- Remote invocation
  - request –reply
  - RPC remote procedure call
  - RMI remote method invocation
- Indirect communication
  - shared memory
  - eventing
  - message queues
- Streams and files
- Note: interaction styles induce some architectural elements
  - e.g RPC infra structure, client / server libraries, message queues

Interaction styles and views

- Interaction styles pertain to communication aspects in the views
  - in contrast to e.g. the subsystem part of the development view, where you find typically *inclusion* and *dependence* relationships
- However, further detailing interaction styles yields new components in all views
  - architectural elements supporting the style
  - often part of a *framework* for the interaction style



Communication as a middleware service



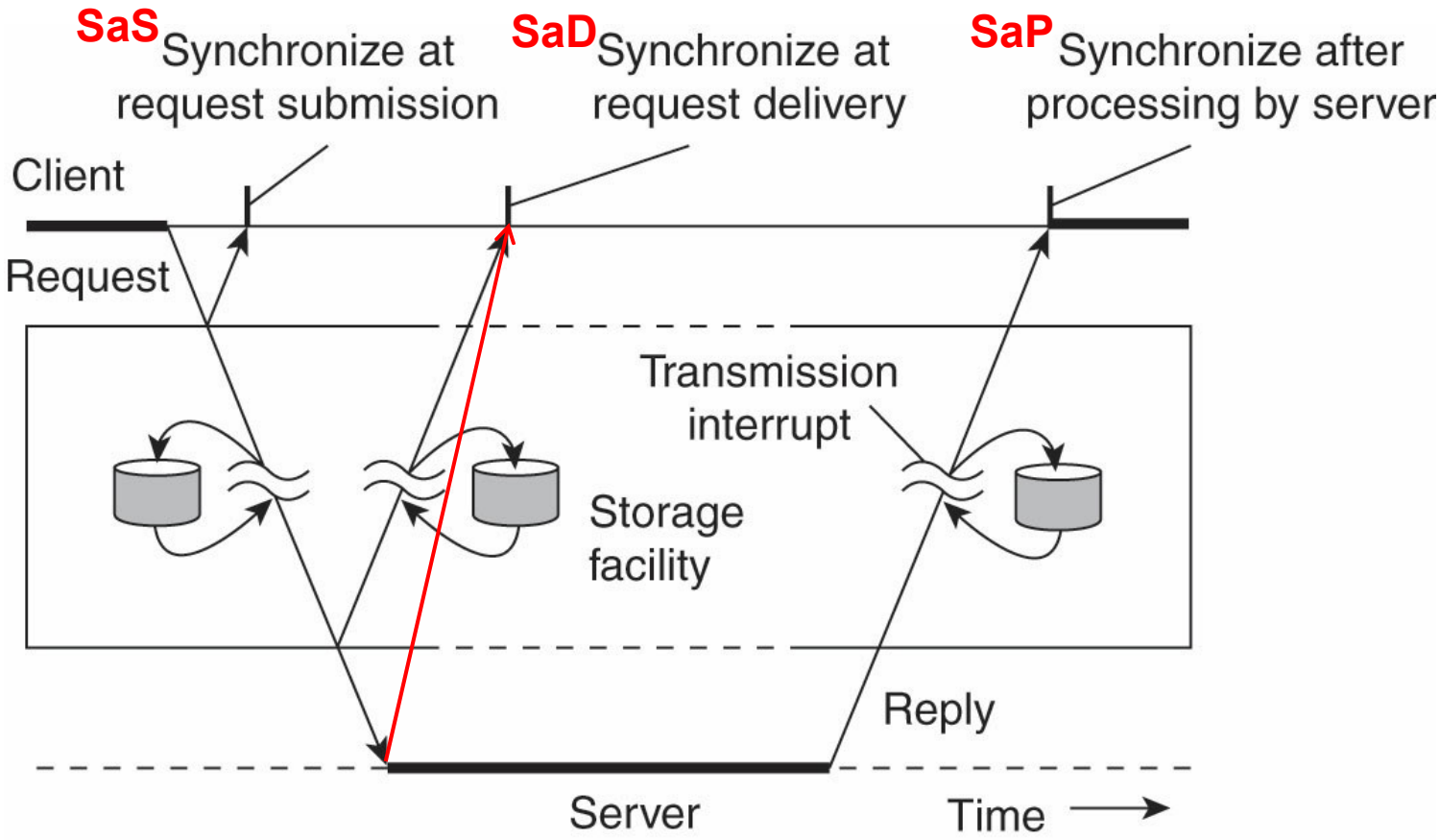
Communicating entities and paradigms

Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

Types (qualities) of communication

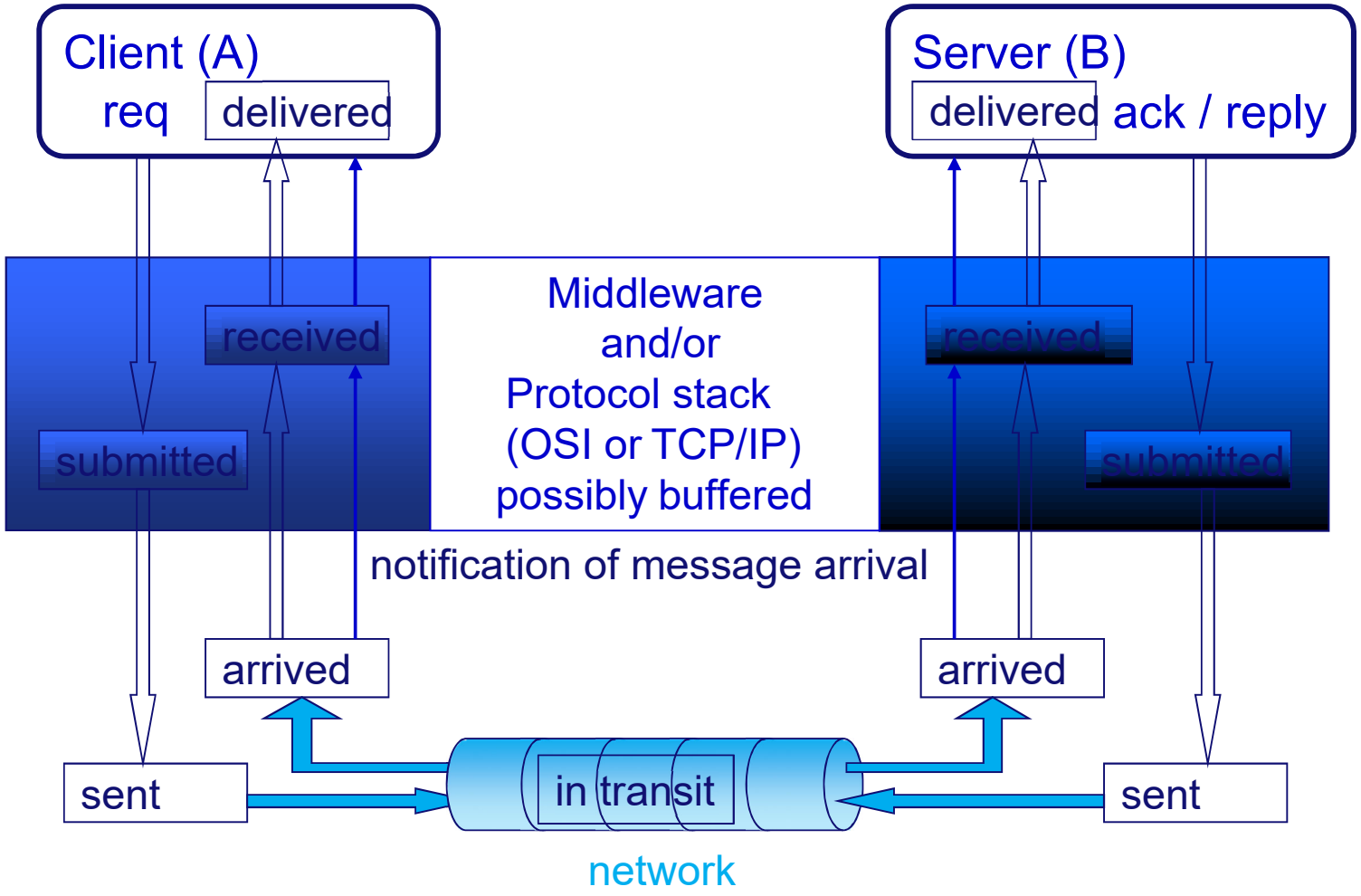
- Memory/storage
  - *transient*: interaction requires sender and receiver to 'execute' at same time
  - *persistent*: interaction (data) remains while sender and receiver disappear
- Synchronization
  - *asynchronous*: sender/caller does not wait or block;
  - *synchronous*: caller blocks till interaction acceptance
    - several different synchronization points, see next slide
  - *buffered*: limited difference between #calls and #responses
- Units of information:
  - *discrete*: structured units, independent and complete
  - *streaming*: basic units; no further communication structure
- Connection
  - Connection-oriented
  - Connection-less
- Reliability
  - Delivery of message guaranteed
- Time dependence
  - temporal relationships, typically with streaming
  - *synchronous*: bounded delay
  - *isochronous*: bound minimum and maximum delay (i.e., jitter)

Synchronization and message storage



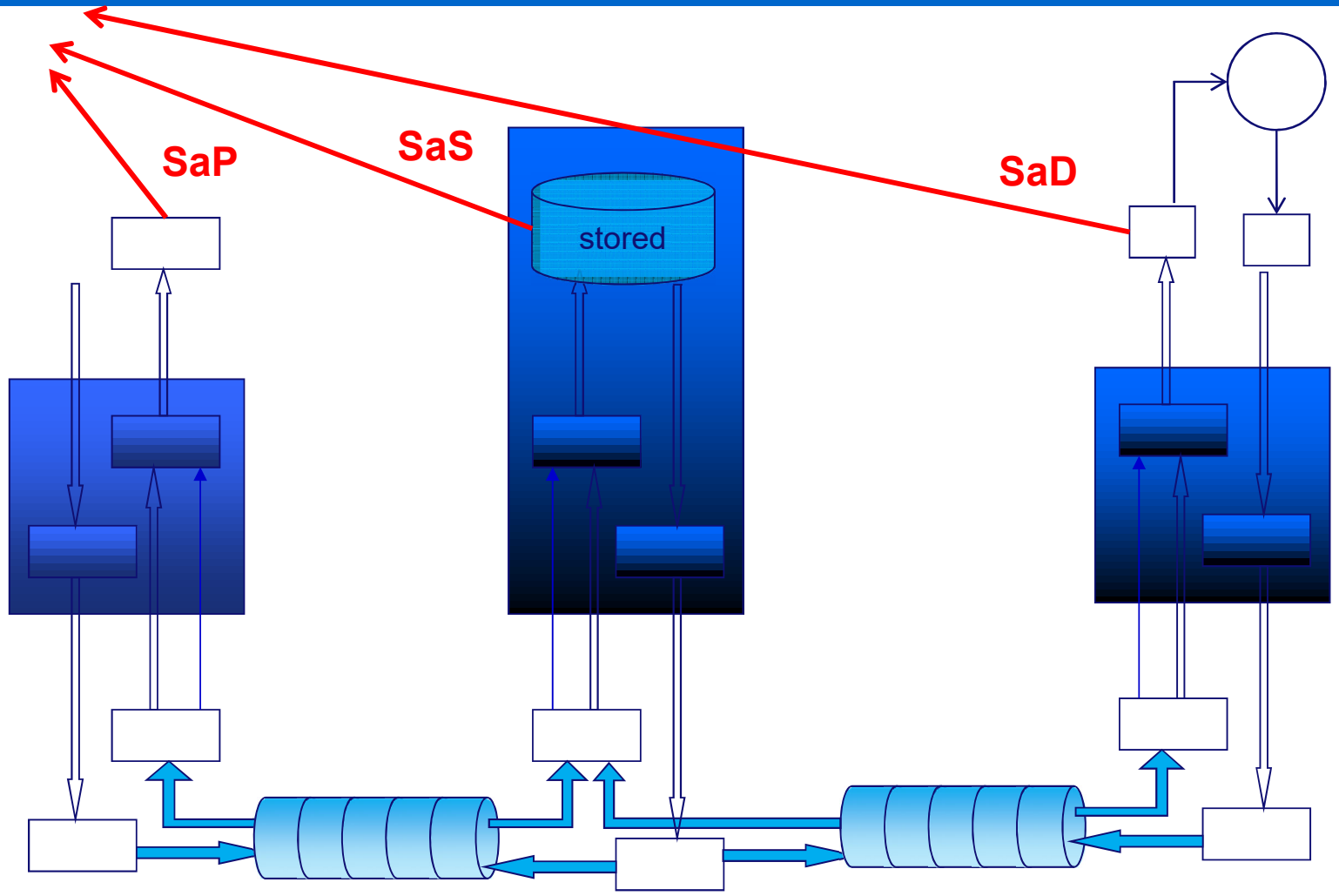
Message status

Naming from the middleware perspective!
Delivered by middleware means received / accepted by application process



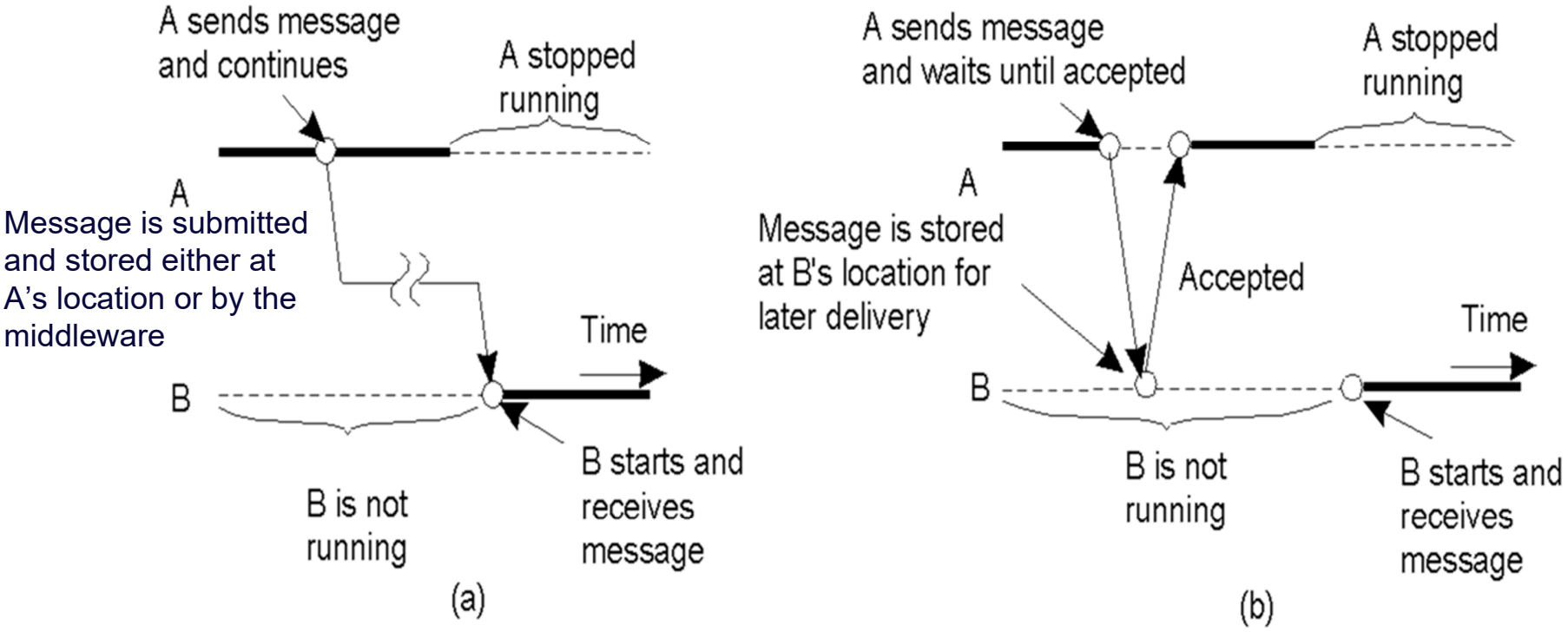
Network with storage

Network with storage

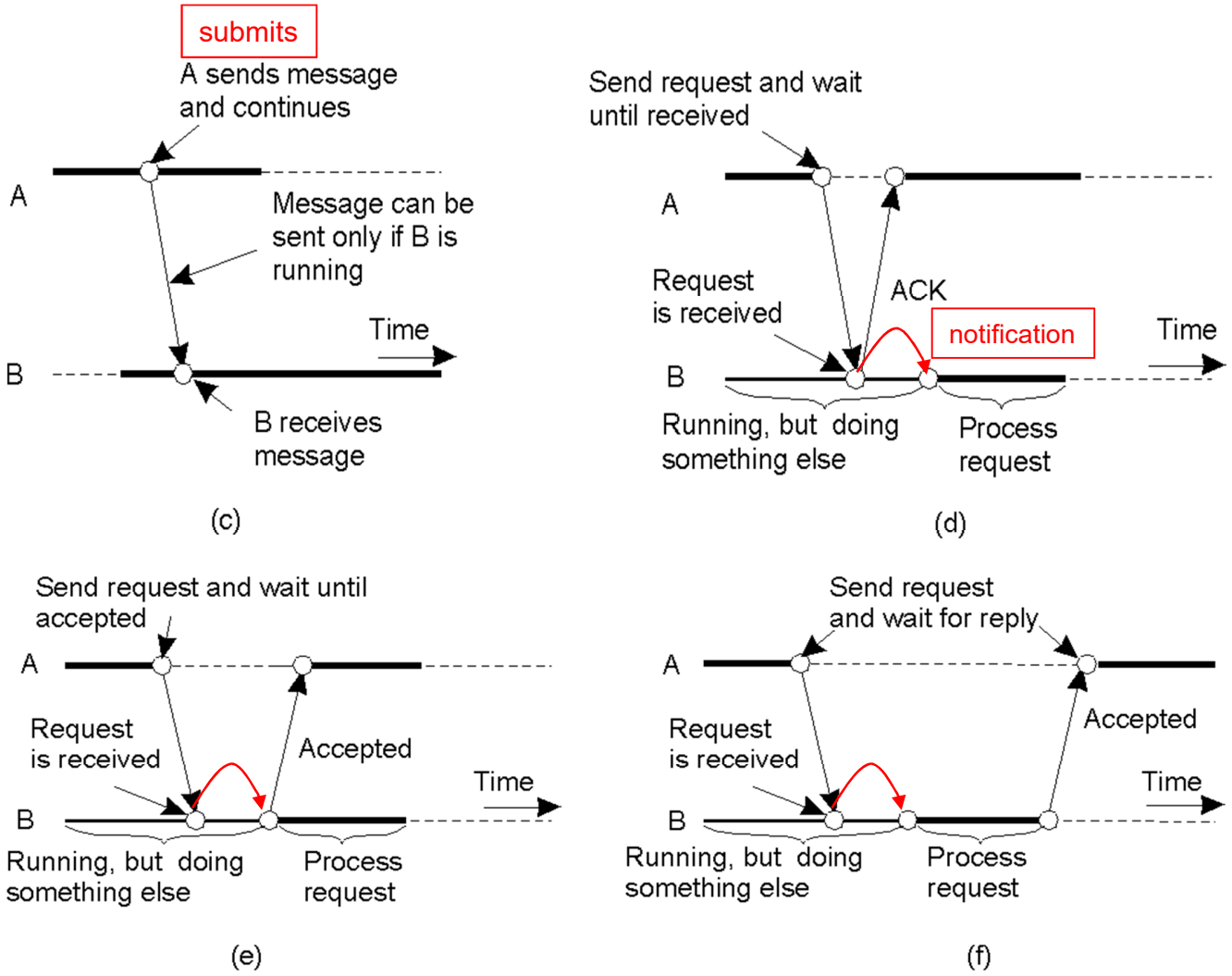


Network with storage

Persistent communication (storage location)



Transient communication



Space and time coupling in distributed systems

	Time-coupled	*Time-uncoupled*
Space coupling Referentially-Coupled (TvS)	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*:
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

Space and time coupling in distributed systems

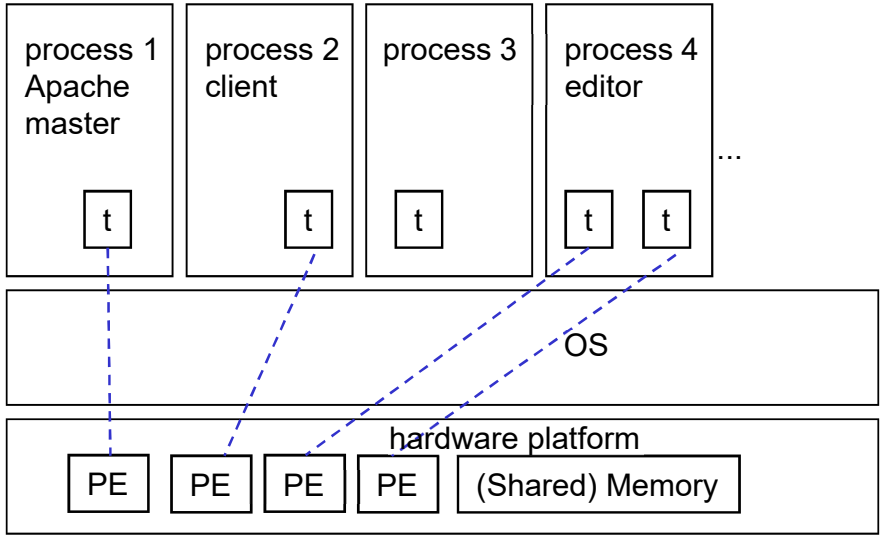
	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: **E-mail**
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

Agenda

- Introduction
- Layered protocols and middleware
- Remote Invocation
  - (Remote) Procedure calls
  - (Remote) Method invocation
- Message Oriented Middleware
- Streaming

Interaction starting point

- **Details of interactions and interacting entities become visible mostly in the process and logical views**
- **Except for regular procedure/method calls, interaction requires independently executing entities (Processing Elements)**
  - processes, threads
    - Communication *on same machine* goes via shared memory (see picture: several processors in the platform execute the processes/threads that share the available memory)
  - processes can execute on different machines
    - Communication *between* machines use a protocol stack
  - also possible for processes on same machine, but via shared memory



Concepts: Process, Thread

- **Process**
  ("program in execution")
  - defines a data space
    - Virtualization of the memory
  - unit of resource management
    - defines ownership of resources
    - concurrency transparency
  - unit of deployment (distribution)
    - together with a software component
  - unit of fault containment
  - has at least one associated thread

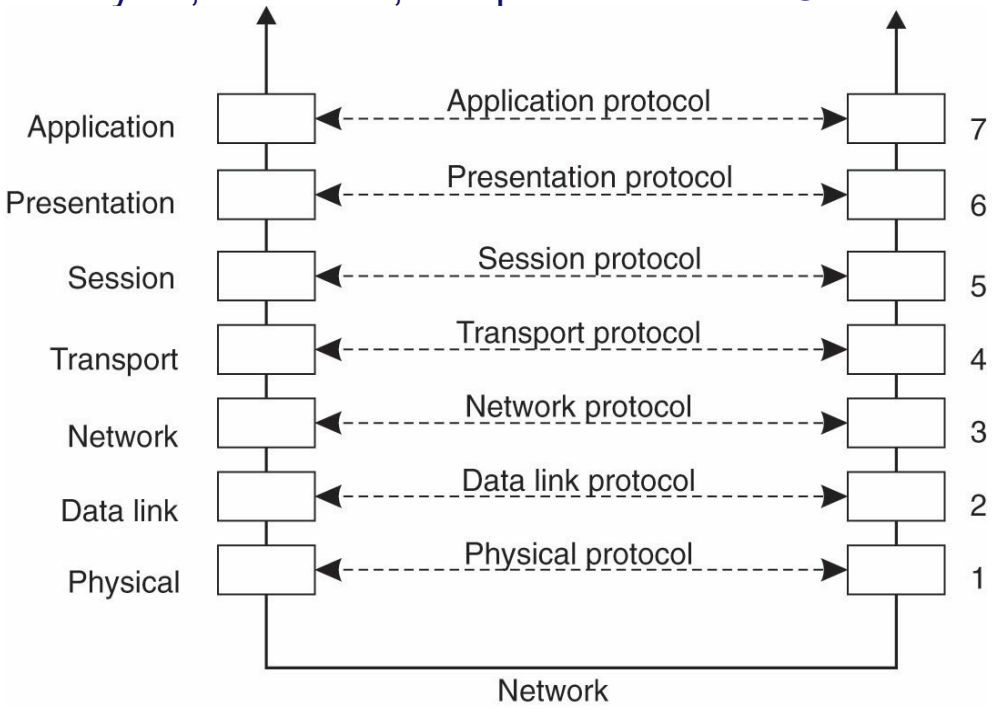
- **Thread**
  - operates in an address space
    - i.e., in a process; several threads may share that space
  - has an associated *execution state*
    - place where it is in the code (PC)
    - stack image and return addresses of function calls,
    - values of processor registers
  - unit of concurrency
    - virtualization of the processor
  - unit of scheduling
    - though in case of one thread per process, the *process* is often said to be the unit of scheduling

Layered protocols: OSI(‘83) reference model

- **Interfaces:**
  - *provided* to layer above
  - *required* from layer below

- **OSI protocols:**
  - are in fact hardly used
  - in particular, session and presentation layer often absent, sometimes a middleware layer takes their place
- **OSI reference:**
  - widely accepted

Layers, interfaces, and protocols in the OSI model.



Although we may think IP is all there was...

Protocol Stacks in Relationship to the OSI Model

OSI Layer	Apple Computer	Banyan Systems	DEC DECnet	IBM SNA	Microsoft Networking	Novell NetWare	TCP/IP Internet	Xerox XNS	OSI Protocols
Application Layer 7	Application Programs and Protocols for file transfer, electronic mail, etc.								
Presentation Layer 6	AppleTalk Filing Protocol (AFP)	Remote Procedural Calls (Net RPC)	Network Management Network Application	Transaction Services Presentation Services	Server Message Block (SMB)	NetWare Core Protocols (NCP	X (Telnet, FTP, SMTP, etc.) Transport Layer Security	Control and Process Interaction	ISO 8823
Session Layer 5	AppleTalk Session Protocol (ASP)		Session	Data Flow Control	Network Basic Input/Output System (NetBIOS)	Network Basic Input/Output System (NetBIOS)			ISO 8327
Transport Layer 4	AppleTalk Transaction Protocol (ATP)	VINES InterProcess Communications (VIPC)	End Communications	Transmission Control	Network Basic Extended User Interface (NetBEUI)	Sequenced Packet Exchange (SPX)	Transmission Control Protocol (TCP), User Datagram Protocol (UDP)	Sequenced Packet Protocol (SPP)	ISO 8073 TP0-4
Network Layer 3	Datagram Delivery Protocol (DDP)	VINES Internet Protocol (VIP)	Routing	Path Control		Internet Packet Exchange (IPX)	Internet Protocol (IP)	Internet Datagram Protocol (IDP)	ISO 8473 (CLNP)
Data Link Layer 2	Network Interface Cards: Ethernet, Token-Ring, ARCNET, StarLAN, LocalTalk, FDDI, ATM, etc. NIC Drivers: Open Datalink Interface (ODI), Network Independent Interface Specification (NDIS)								
Physical Layer 1	Transmission Media: Twisted Pair, Coax, Fiber Optic, Wireless Media, etc.								

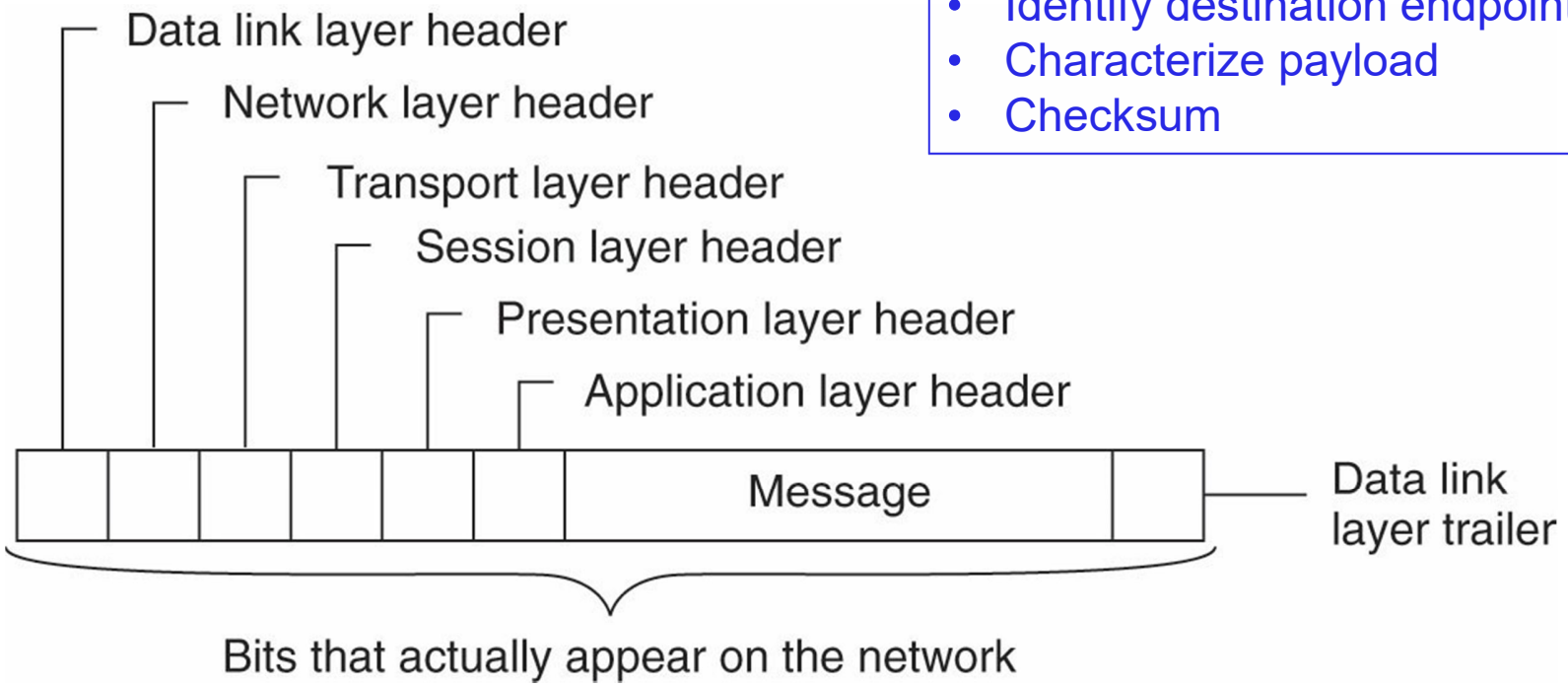
See also https://en.wikipedia.org/wiki/List_of_network_protocols_(OSI_model)#O

Issues addressed by the layers

- **Basic network protocols:**
  - Physical:
    - sends *bits* on medium (i.e. standardizes the electrical, mechanical, and signalling interfaces)
  - Data link:
    - detects and corrects errors in *frames*; deliver frames in one-hop neighbourhood
  - Network:
    - send *packets* from sender to receiver machines using multi-hop routing
  - Transport:
    - breaks *messages* into packets; delivery guarantees; multiplexing ports
- **Higher-level protocols:**
  - Session:
    - provides dialog control and synchronization facilities (checkpoints);
  - Presentation:
    - structures information and attaches meaning ("semantics": e.g. names, addresses, amount of money);
  - Application:
    - network applications, including a collection of "standard" ones (email, file transfer)

Layered protocols and message layout

Headers:
- Identify destination endpoint
- Characterize payload
- Checksum

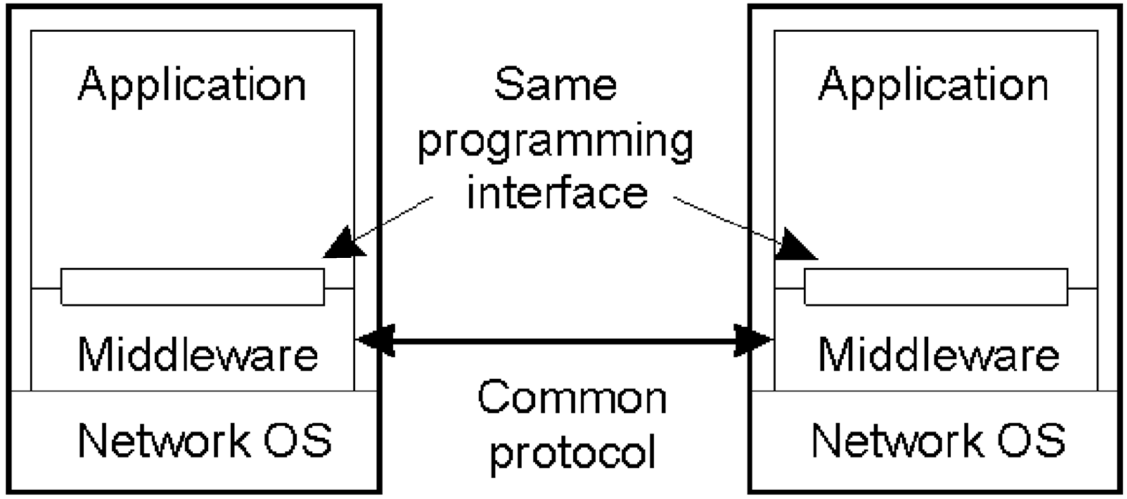


A typical message as it appears on the network. Note that only link layer messages ever "exist"

Layering and middleware

- **Layers >4 build services on top of transport facilities**
- **A general definition of such services is *middleware***
  - going beyond simple transport / message passing
    - RFC 2768 mentions: remote execution, data management, resource management, security, mobility support, ...
  - therefore, the focus shifts from the protocol definition to the programming interface (provides uniform computation model)
  - middleware also refers to a layer that hides distribution aspects

- **Programming interface: API for this service**
- **Network OS: delivers transport service**

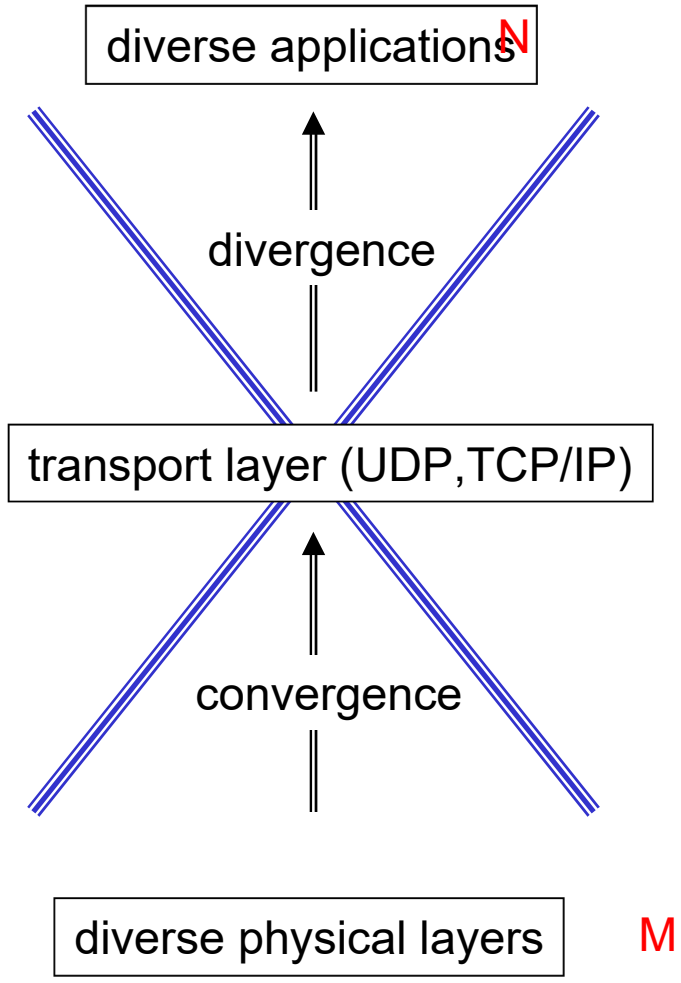


Middleware: the hour glass problem

- IP/everything tendency
  - goal in the layering: connect everything N x M
    - UDP, TCP/IP as basic interoperability N+M

- Application needs are diverse
  - use transport layer for private purpose
  - no direct need to support standards
  - hence the problem: difficult to define generally accepted middleware layer
  - result: middleware libraries for many application domains

- Interaction style frameworks typically come with a middleware library (e.g.

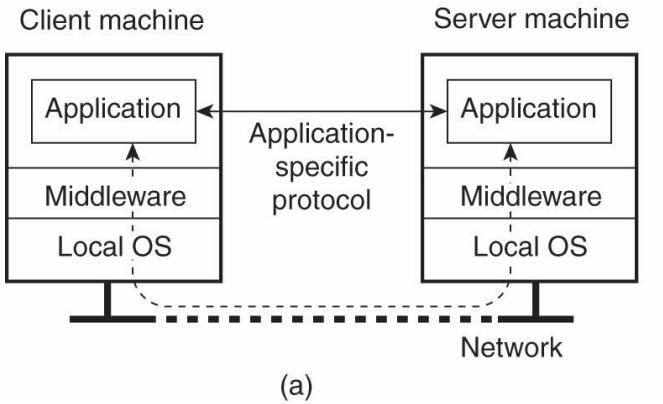


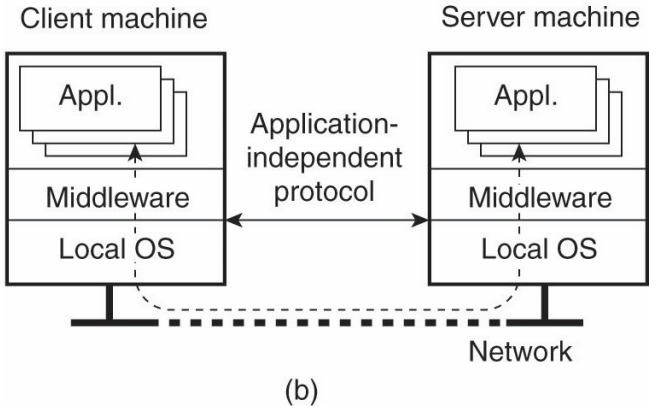
Middleware: the hour glass problem

(CORBA, UPnP)

Middleware tradeoff

- Two interfaces:
  - Application ↔ middleware
  - network protocol between devices
- Case (a): middleware merely provides message passing
  - using just IP-convergence
  - each application maintains its own specific protocol
  - no sharing of logic
- Case (b): middleware provides advanced services
  - services define the (application independent) protocol between middleware layers
  - sharing of standard functions
    - supports the thin-client approach





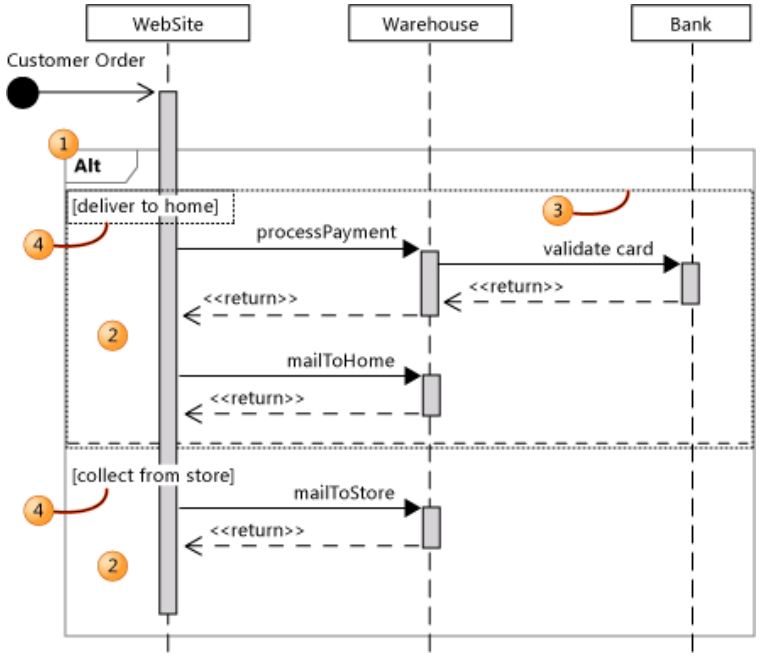
Agenda

- Introduction
- Layered protocols and middleware
- Remote Invocation
  - (Remote) Procedure calls
  - (Remote) Method invocation
- Message Oriented Middleware
- Streaming

Procedure/method call

- Called function runs on ‘thread of control’ of caller
  - synchronous, discrete, most often transient (, connection-less)
  - synchronous call (with completion return) or signal (without return)
- This is the normal ‘mode’ of operation in regular programs (processes)
  - call to functions in same process, including libraries
  - implied support: call stack
- Extended to interaction across process and machine boundaries: *remote procedure call*
  - visible in the process and deployment views
  - implied support: infra structure for
    - establishing the remote call and return
    - binding the RPC reference to the function to be called on the remote machine

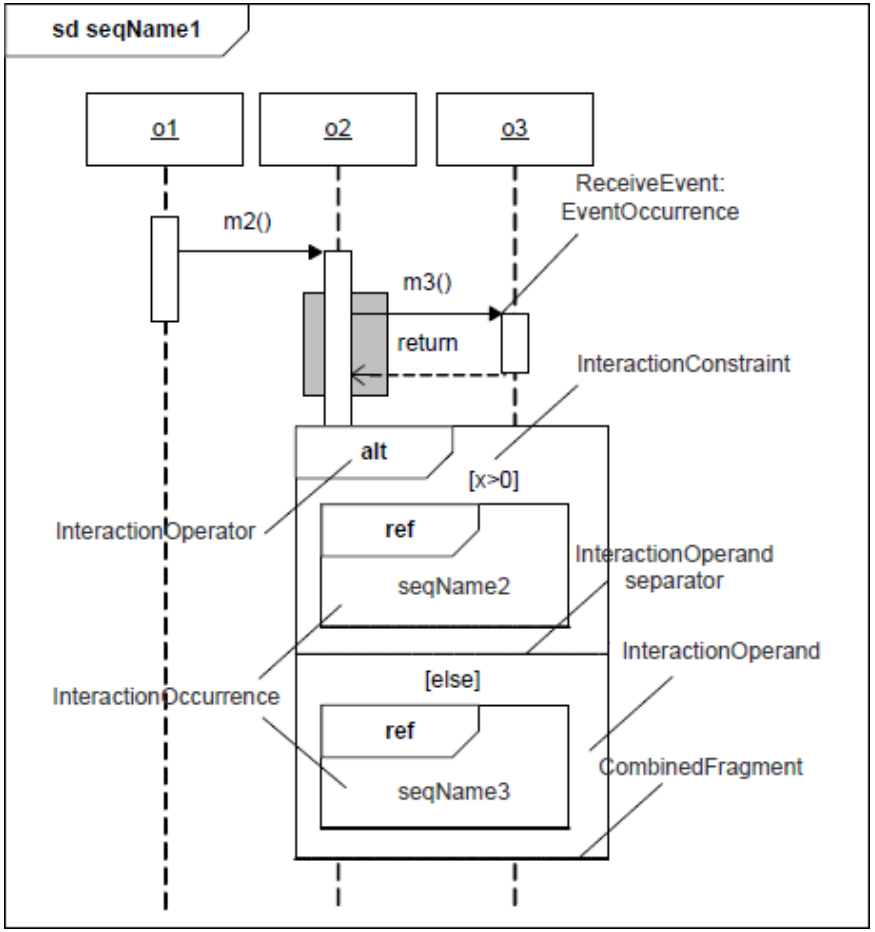
Process view: (Remote) procedure calls



from: http://msdn.microsoft.com/en-us/library/dd465153.aspx

Numbers refer to MS tool description

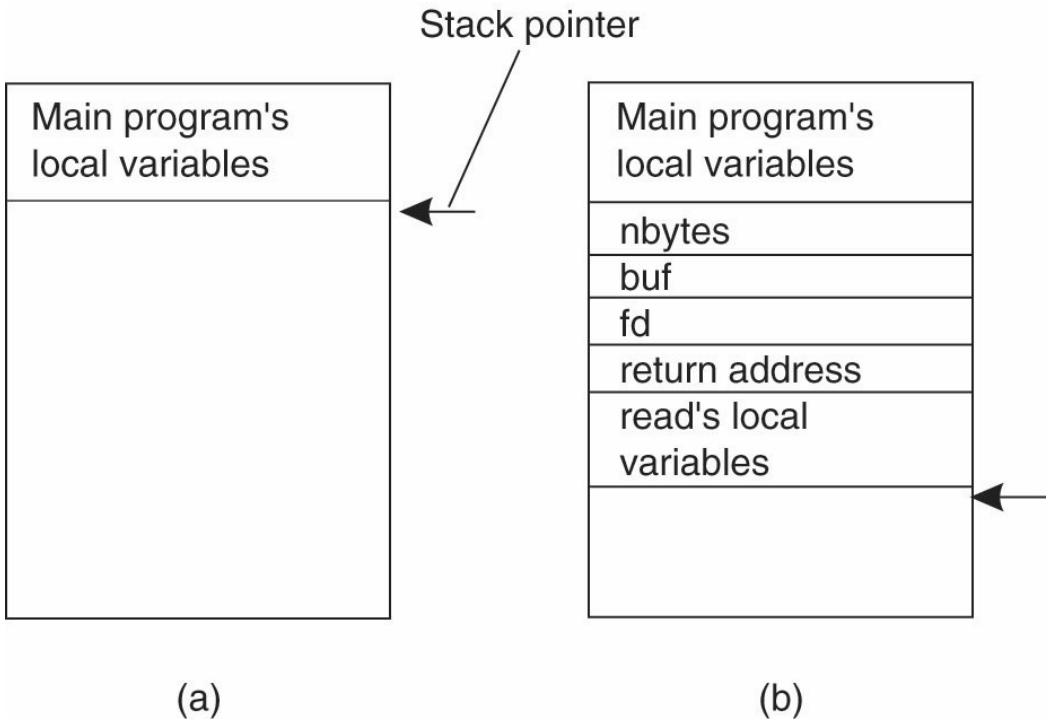
1. combined fragment (alt)
2. interaction operand
3. boundary
4. guard



from: Control Flow Analysis of UML 2.0 Sequence Diagrams, Vahid Garousi, Lionel Briand and Yvan Labiche

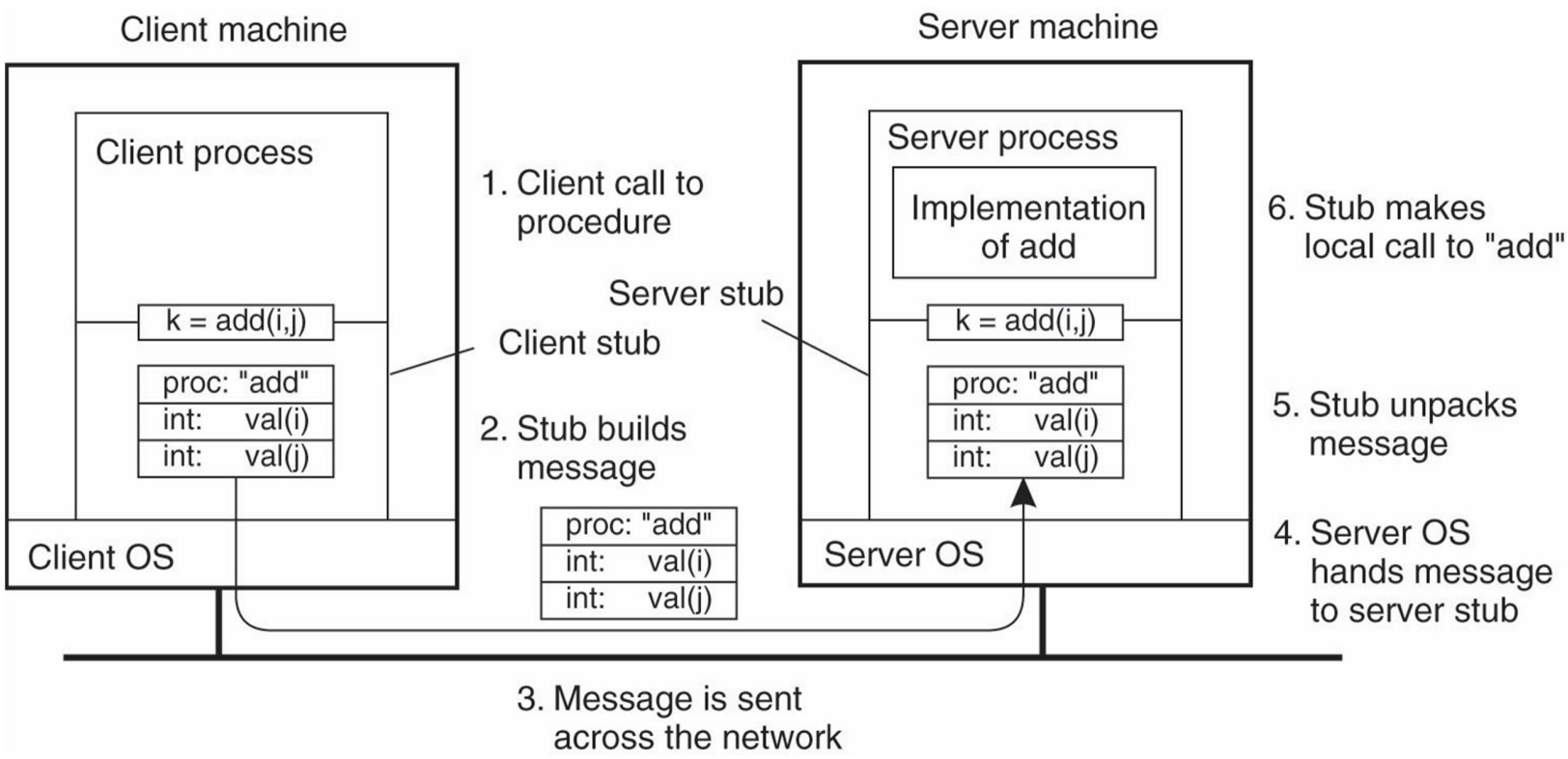
Conventional procedure call

Call: read(*int* fd, *unsigned char* *buf, *int* nbytes)



(a) Parameter passing in a local procedure call: the stack before the call to *read*.
(b) The stack while the called procedure is active.

Steps in calling an RPC



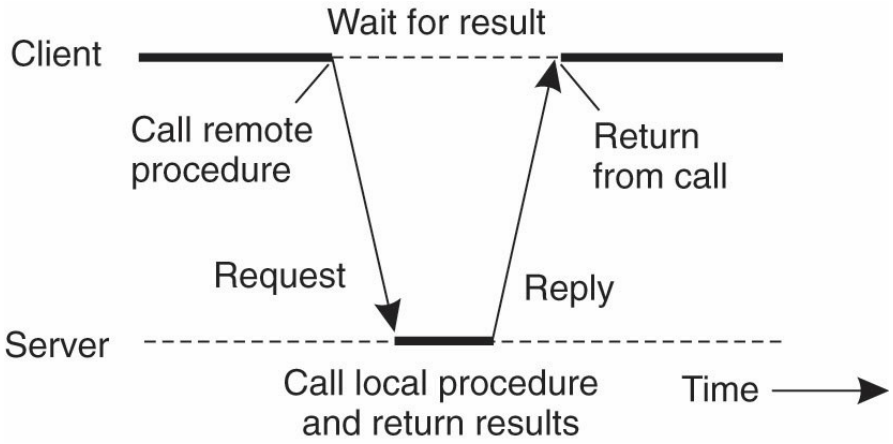
The client stub must be aware of, or discover, the location of the server.

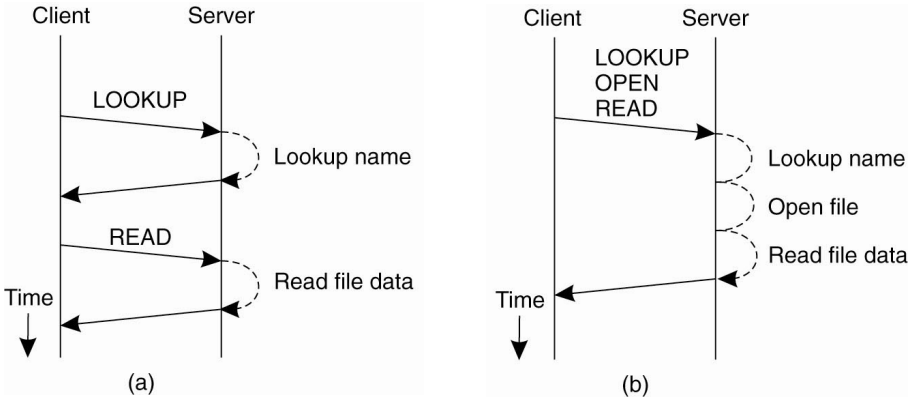
Steps in calling an RPC

1. Client procedure calls client stub in *the 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. Client *stub* unpacks result, returns to client

Remote procedure call

- Typical in Client-Server style
- The concept aims at
  - *access transparency*
    - provide a *procedural interface* to remote functionality
    - difference local/remote not visible
    - .....except for non-local resources
      - typically, remote data
  - *portability* (of existing code)
  - reduce language/OS *dependence*
- Concerns of
  - scalability: many clients calling
  - reliability: increased dependence
    - independent failure of the server
      calls for an elaborate solution
  - performance, see picture





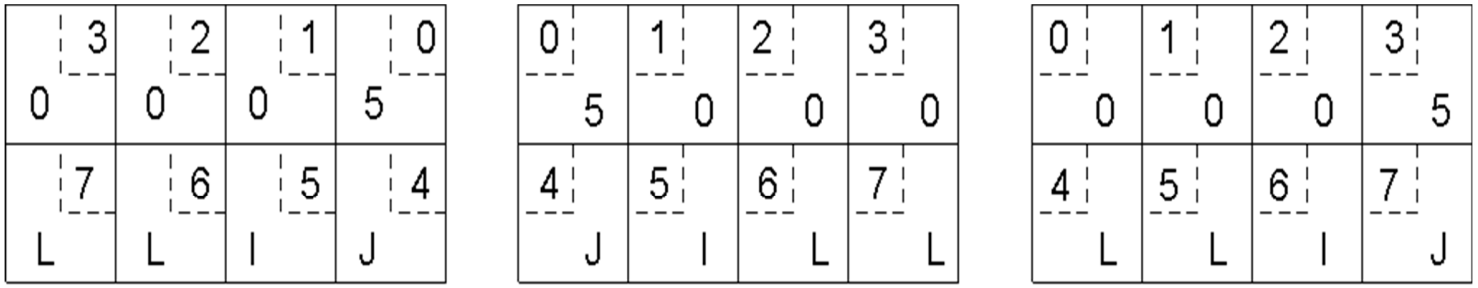
(b) combines several calls of (a)

Elements of a realization

- *Marshalling* of data
- Definition of the *RPC protocol*
- How to deal with *parameters*
- The development process to *include RPCs into a program*
  - frameworks
- RPC server infra structure, server *discovery, binding time*
  - again, a framework
- Semantics under partial failures
- Synchronization (alternatives)

Need for marshalling

- Representations of numbers, characters, and other data items on machines may differ
  - The little numbers in boxes indicate the address of each byte



(a) (b) (c)

a) Original message on the Pentium (“little endian”)
b) The message after receipt on the SPARC (“big endian”)
c) The message after simple inversion is still wrong

Hence, information about the meaning of the bytes in a message is needed.

- can be provided by a schema that is included in the representation

Marshalling versus serialization

Marshalling: the activity by which a stub converts local application data into network data (eXternal Data Representation) and packages the network data into packets for transmission.

Serialization: the activity by which the state of an object is converted into a byte stream in such a way that the byte stream can be converted back into a copy of the object.

- compaction schemes are important for saving memory and network bandwidth/latency

Difference becomes noticeable for objects.

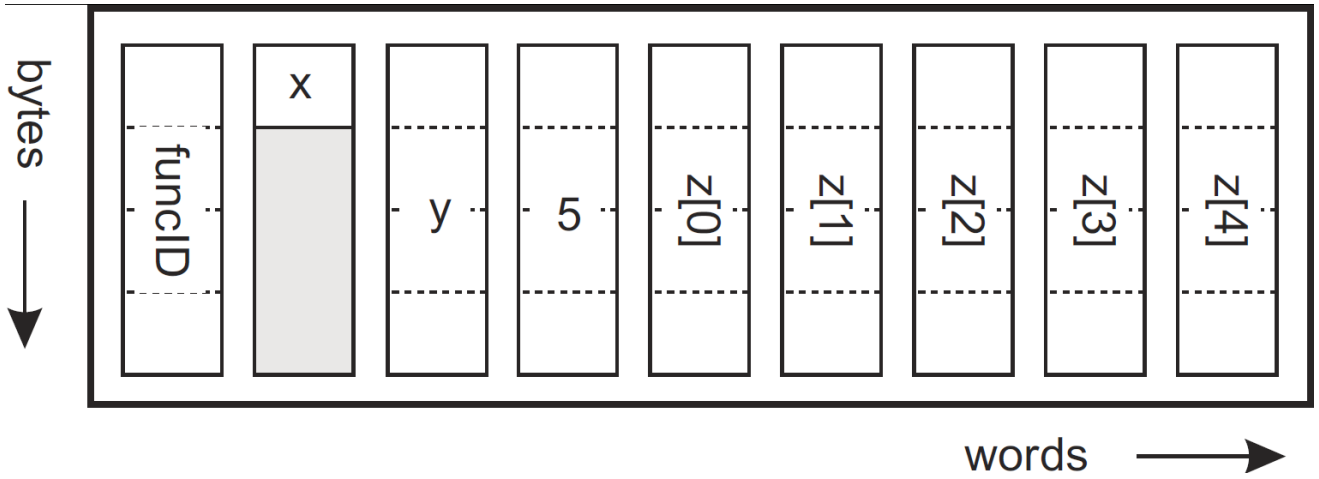
- objects have a code base that also needs to be marshalled.
- Java serialization relies on the codebase being present at the destination
- there is a need to deal with references
- recursive and shared objects must be handled efficiently
- entire trees, based at root-object, are serialized,

In Python (*pickle module*) these activities are considered the same, in Java (class *implements java.io.Serializable*) not.

Protocol

- Message format and data structure representation, e.g.
  - integers in two's complement, chars in 16-bit Unicode, etc.
  - marshalling (conversion to an external data representation)

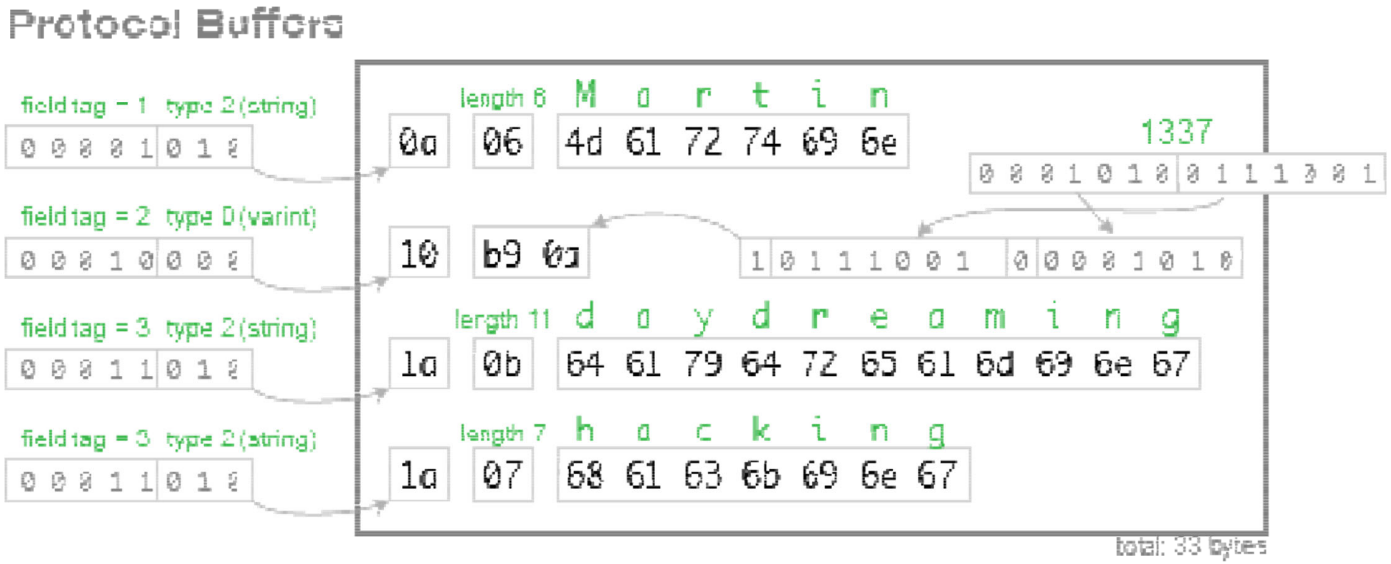
```
void someFunction(char x; float y; int z[5])
```



- Message exchange sequence
  - e.g. acknowledge
- Binding to carrier
  - TCP, UDP, ...

Protocol buffers: xdr-format developed by Google

```
message Person {
        required string user_name = 1;
        optional int64 favourite_number = 2;
        repeated string interests = 3;
}
```



Taken from:

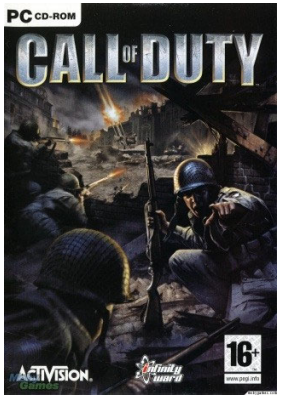
https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html

RPC parameter passing

- Usually, value/result semantics
  - no reference parameters, no statics (globals)
  - ...lack of transparency
- Reference parameters:
  - address space problem
  - copy/restore
    - just copy for in- and restore for out-parameters
  - remote reference mechanism
    - …additional RPC(s) for complex data structures?
    - …migrate *code* towards *data* rather than using RPCs?
    - what to do with concurrency?
- For RPCs to be useful remote resources are typically required
  - e.g. implicit state (file server), through library calls, or global references (URI)

Example: DCE RPC framework

- Distributed Computing Environment RPC
  - original design by the *Open Software Foundation* later that became the *Open Group*
  - MS implementation in Exchange/Outlook: MS-RPC
  - Basis for DCOM and used in Samba (fileserver)
  - supposedly part of *Call of Duty,* multiplayer game

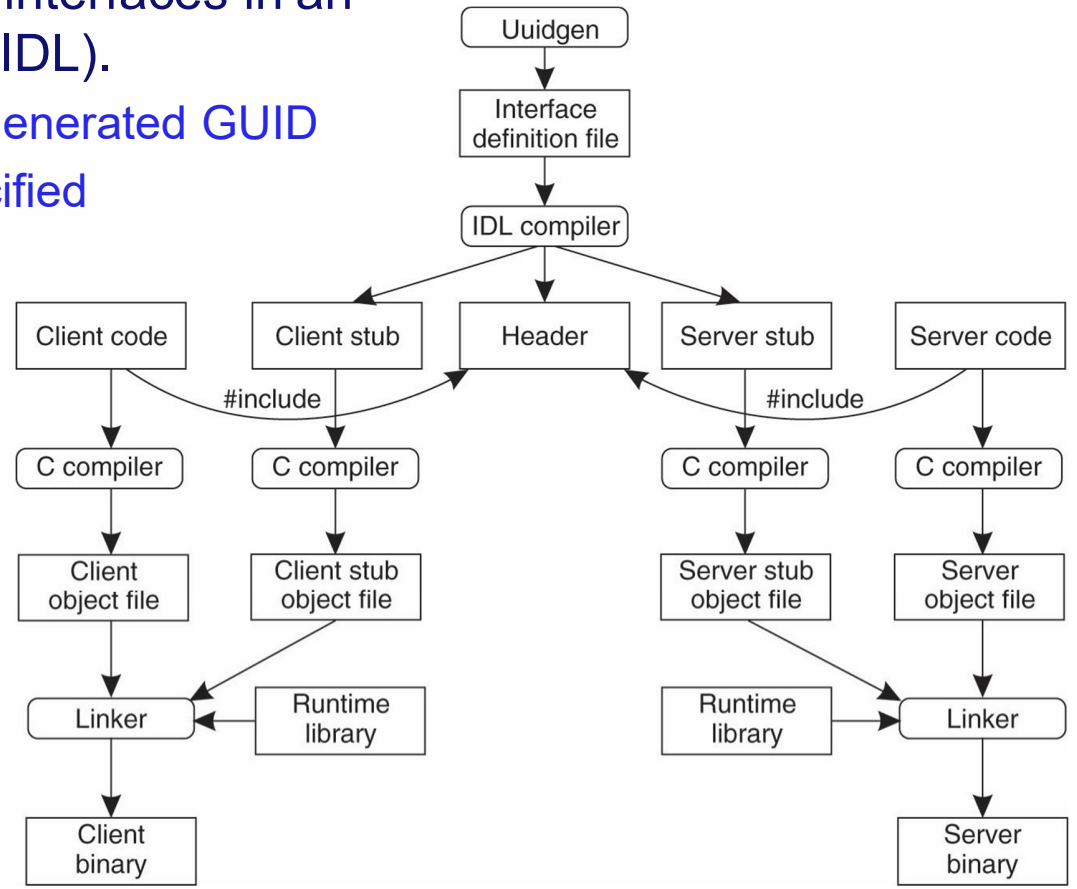


- DCE Framework
  - tooling for supporting RPC-based client & server design
  - provided services
    - DCE/RPC (remote procedure calls),
    - DCE/DFS (distributed filesystem)
    - timing service
    - directory service (for lookup and discovery)
    - authentication service

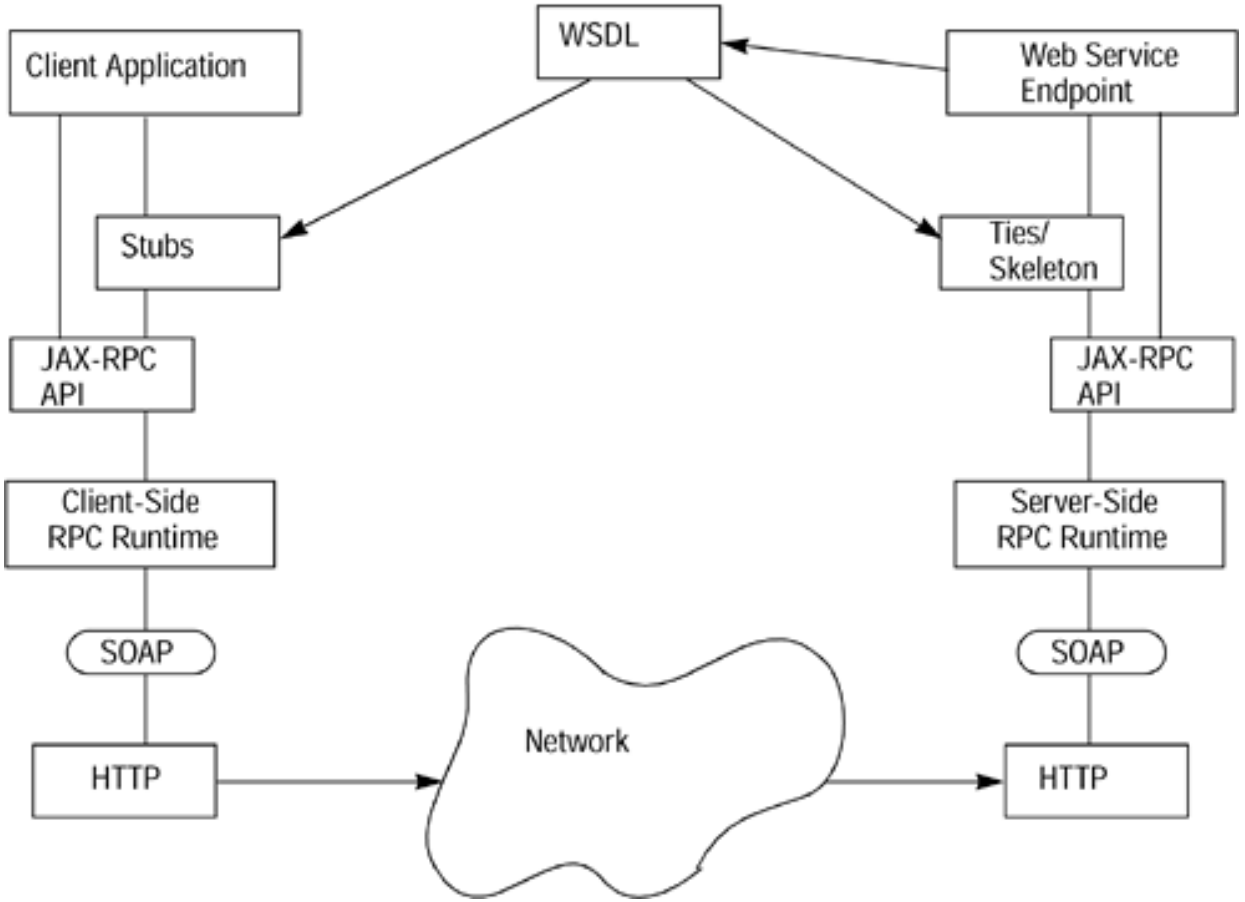
- Other RPC/ Serialization frameworks: gRPC, Apache-thrift, Avro, …

Developing with DCE RPC

- Developer specifies the used interfaces in an interface definition language (IDL).
  - uniqueness guaranteed by a generated GUID
  - IDL datatypes have XDR specified
- An IDL compiler generates
  - a common include file
  - client and server stubs (code)
- Developer fills in the code for the functions in the IDL file
- The linker combines the resulting object file with stub and libraries



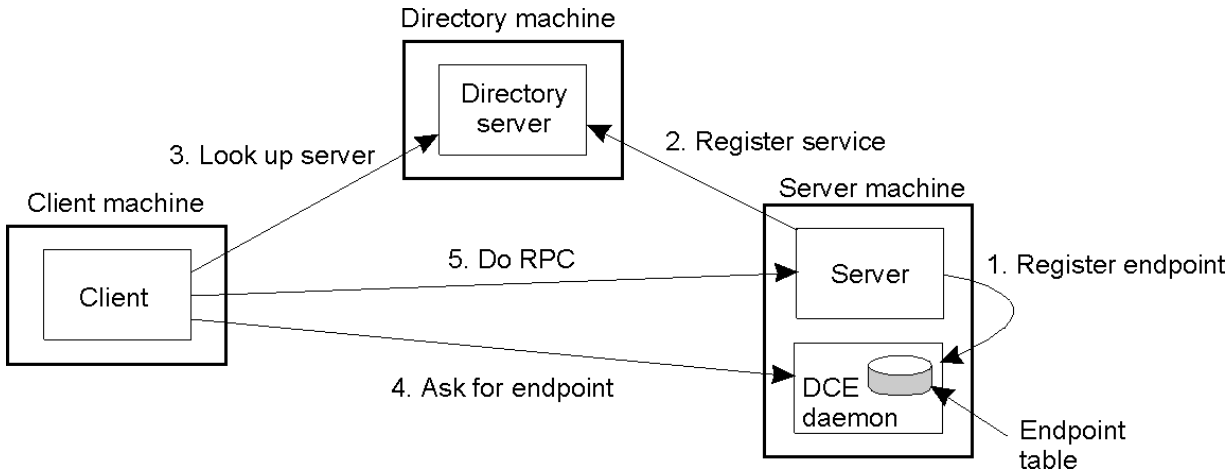
JAX-RPC for web services



Taken from: http://www.xyzws.com/scdjws/SGS34/2

DCE runtime services: discovery, registration

- *Server machines*
  - run a ‘daemon’
- A *directory machine*
  - runs a *directory service that* stores (service, server)-pairs



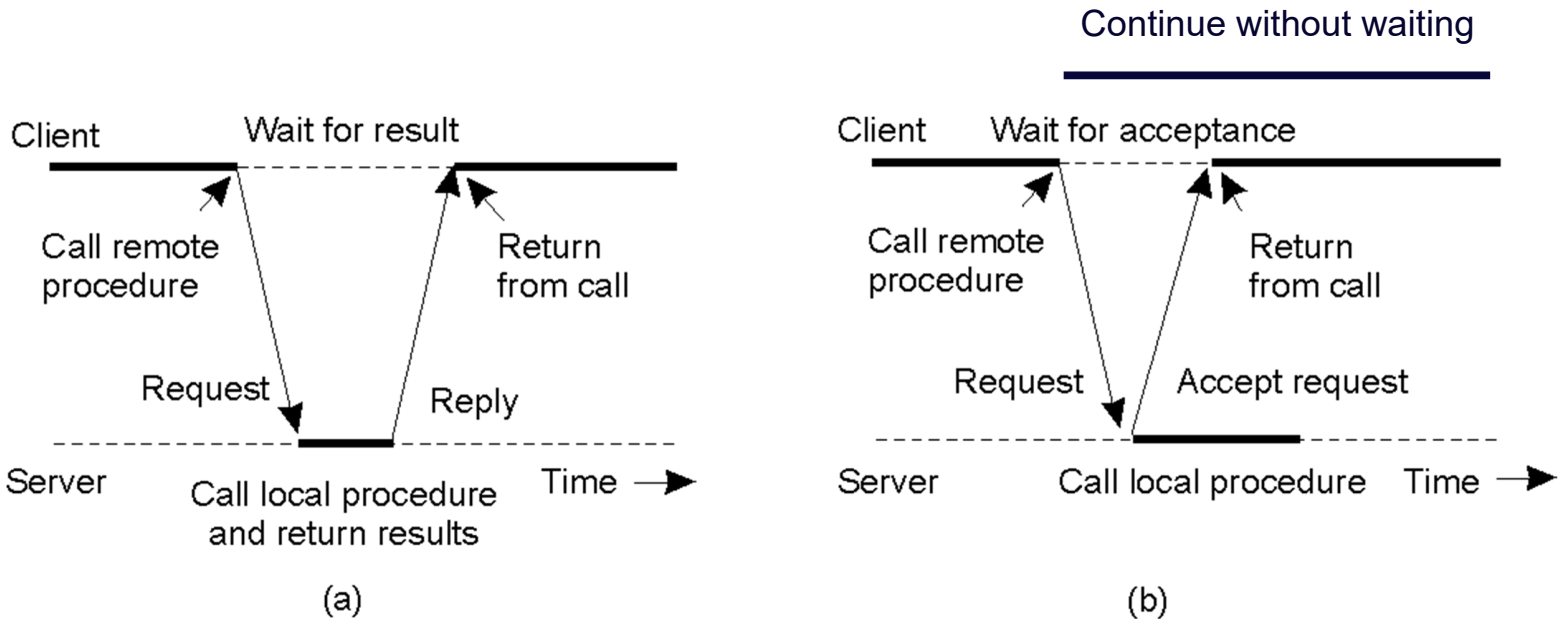
  - A server process:
    - registers itself with the daemon and obtains an endpoint (a ‘port’)
    - registers itself together with the identity of the machine on which it runs with the directory
  - A client process:
    - finds an appropriate server machine
    - finds the server endpoint through the daemon
    - calls the RPC

Partial failure of RPC

- Important starting point: *make a failure model*
  - describe the behavior against which resilience is required
    - part of the problem specification, viz., the environment
- Failure model:
  - request loss, duplication
  - server crash
  - server response loss, duplication
  - client crash
    - may leave orphan tasks
- Responses to these failures:
  - detection via timeout and explicit acknowledgement
  - make requests *idempotent*
    - multiple issue is identical to single issue
    - timestamping, dividing time into *epochs* (to indicate server restarts)
  - define semantics explicitly
    - at most once (DCE), at least once, (exactly once is ideal but unachievable)
  - use timeout mechanism and ‘clean start’ for removing orphans

Asynchronous RPC

- No need to wait if there is no result

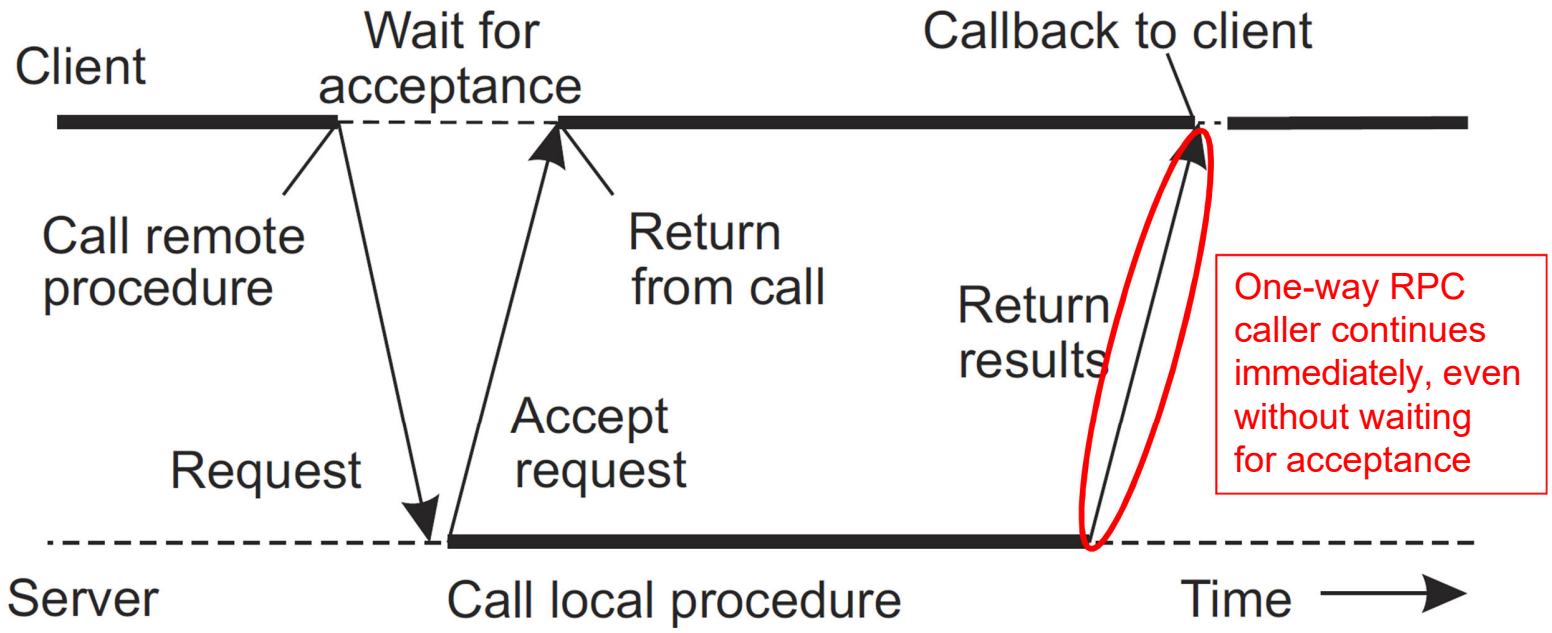


(a) Traditional RPC and (b) Asynchronous RPC

- Variant: one-way RPC
  - do not even wait for acknowledgement of request
  - hence cannot rely on the request being delivered (and processed)

Deferred Synchronous RPC

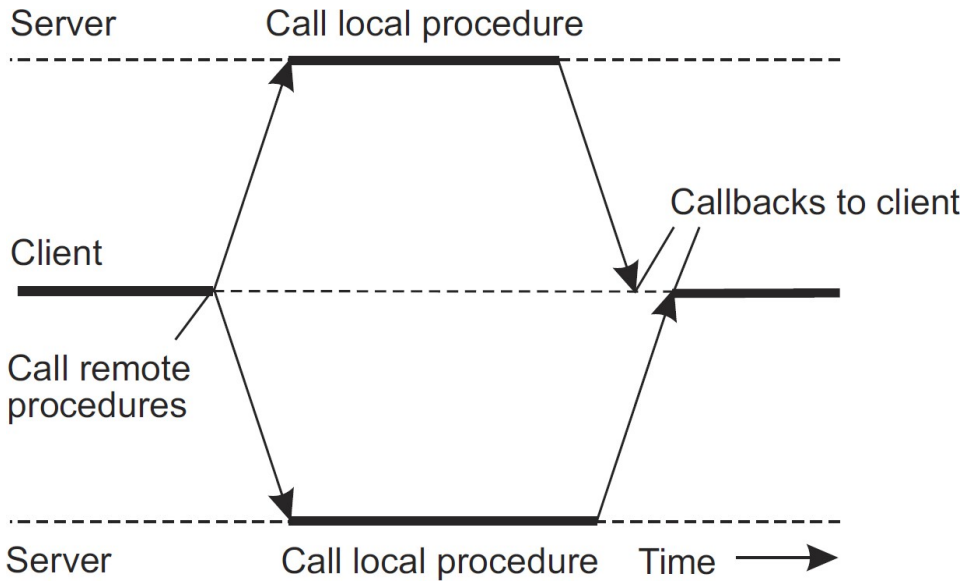
- Do something useful while waiting
- two asynchronous RPCs, 2nd possibly without result ('one-way')



A client and server interacting through two asynchronous RPCs.

Multicast RPC

- Combination of deferred and one-way RPC
- Client request consists of multiple one-way RPCs
  - client need not be aware of this
- Servers concurrently process requests and all reply with a callback
  - client need not consider all responses, e.g., only the first



Agenda

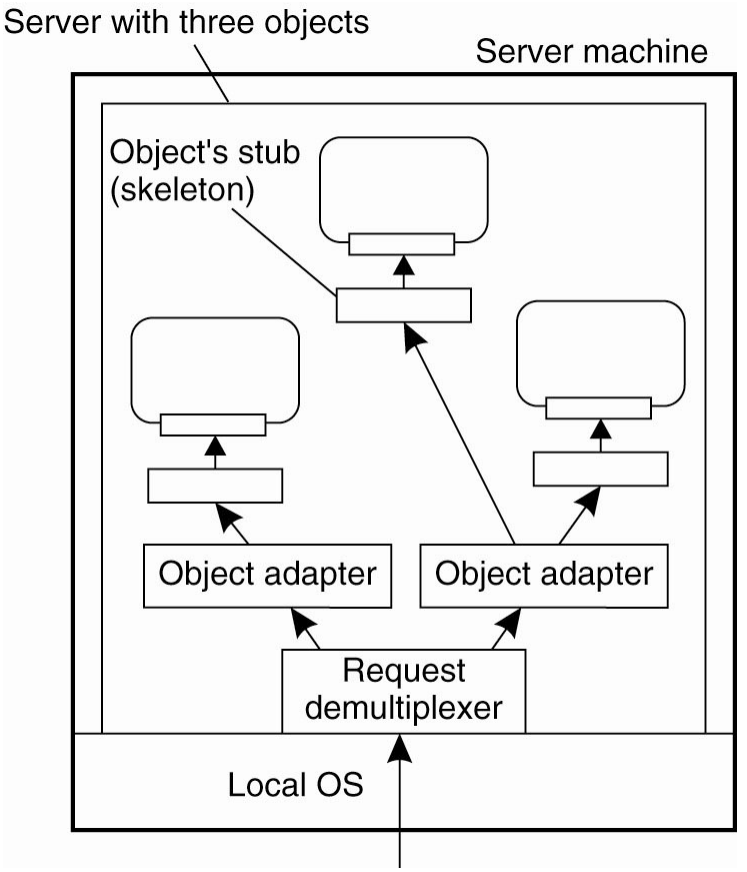
- Introduction
- Layered protocols and middleware
- Remote Invocation
  - (Remote) Procedure calls
  - (Remote) Method invocation
- Message Oriented Middleware
- Streaming

Plumbing elements: proxy, stub, skeleton

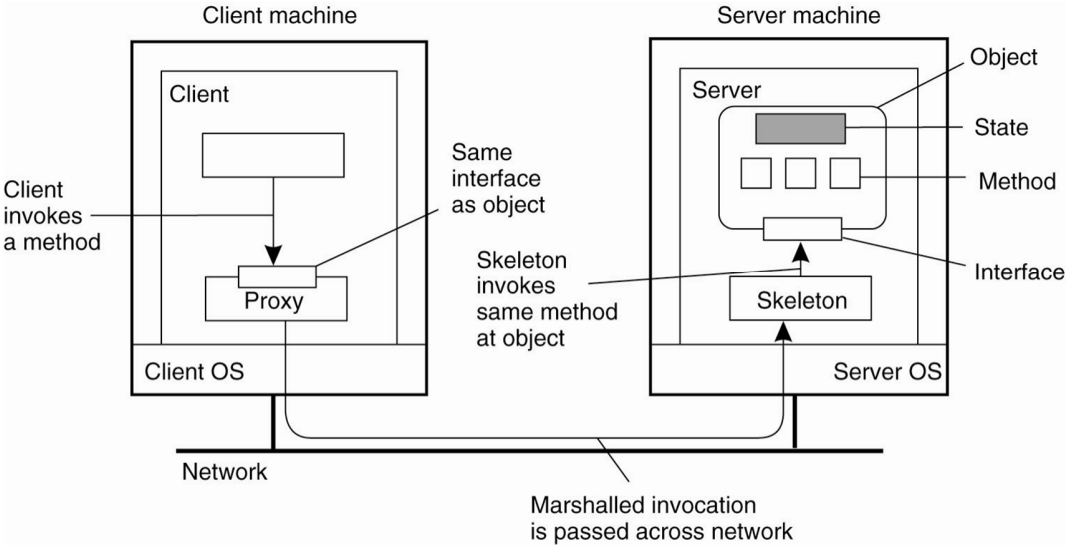
- Proxy:
  - a component that acts on behalf of another component, implementing the same interface
  - capable of implementing filtering policies (e.g. which requests to pass on) and sometimes caching
  - typically, 'proxy' refers to a client side entity (representing the server); a '*reverse proxy*' is placed at a server side
- Stub:
  - (originally): an empty interface implementation
  - (RPC/RMI): client stub: transparently implements an interface for a remote object; responsible for the messaging (also called: proxy) server stub (or skeleton): transparently perform the calls of a *client stub* and handles the messaging
- Example:
  - a HTTP proxy server acts on behalf of a user and filters requests
    - i.e., it is a client to an origin server

Plumbing elements: adapter, broker

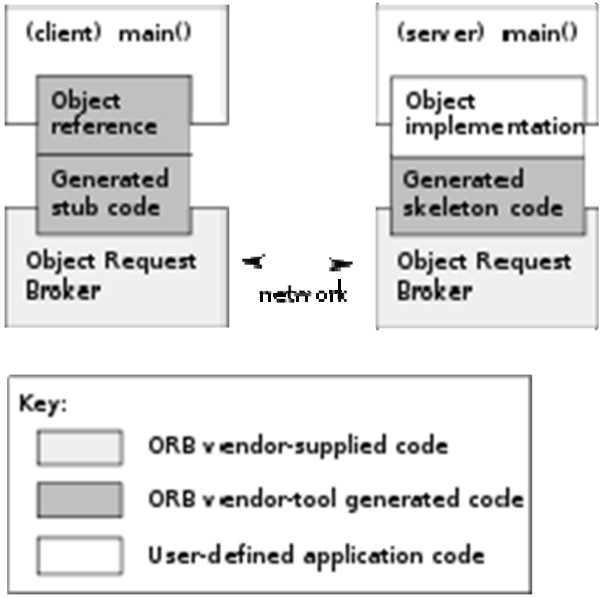
- Object Adapter or Object Wrapper:
  - a component that relays calls to an object interface and manages it
    - typically implementing different management policies for the object, e.g. creation policy, multi-threading, perhaps transient/persistent
    - converting between interfaces
    - possibly state holding for that conversion
- Broker
  - a component that handles and translates calls (messages) between two or more parties, and that manages the binding between references and objects.
    - This binding can be dynamic, based on *interface inspection.*
    - this dynamic binding becomes visible by explicit invocation (instead of a transparent method call) "*RMI (object ref, method name, parameters)*"



Distributed objects



General organization as typically supported by a framework. The *proxy* can be generalized to a *broker* as in the CORBA case.



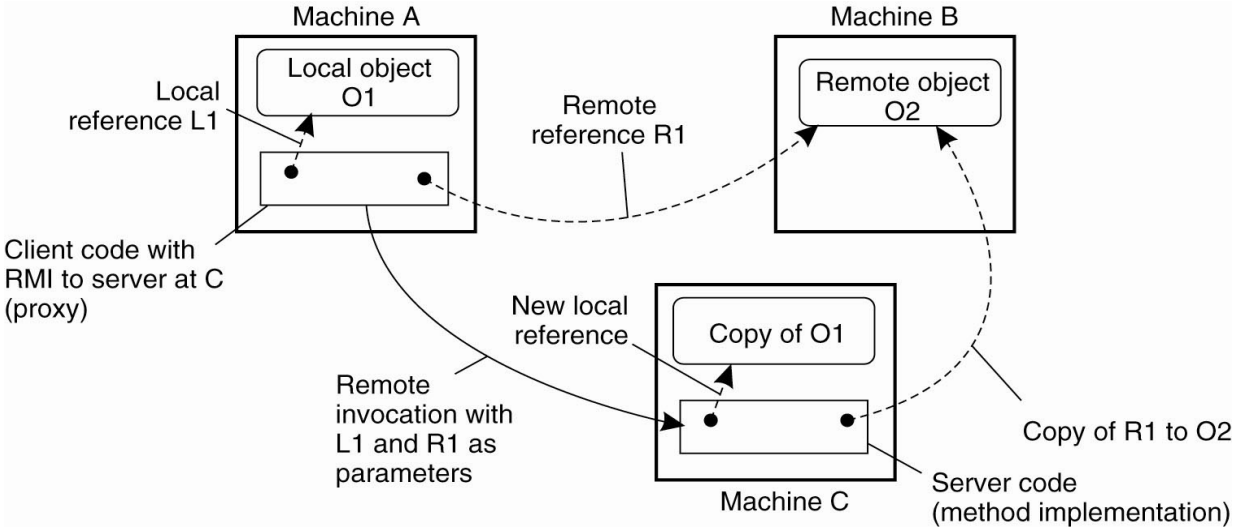
CORBA organization based on an IDL (simplified picture, without services)
Communication is by the IIOP, Internet Inter-ORB Protocol

picture from wikipedia

Remote object invocation and parameters

- RMI: very similar to RPC, difference:
  - an object represents a context (a state) together with operations on it
    - objects can thus be persistent or transient
  - clear separation between interface and implementation (~ SOA)
    - actual implementation is hidden, can be a program in any language (servant in CORBA)
  - references to objects occur naturally in a program

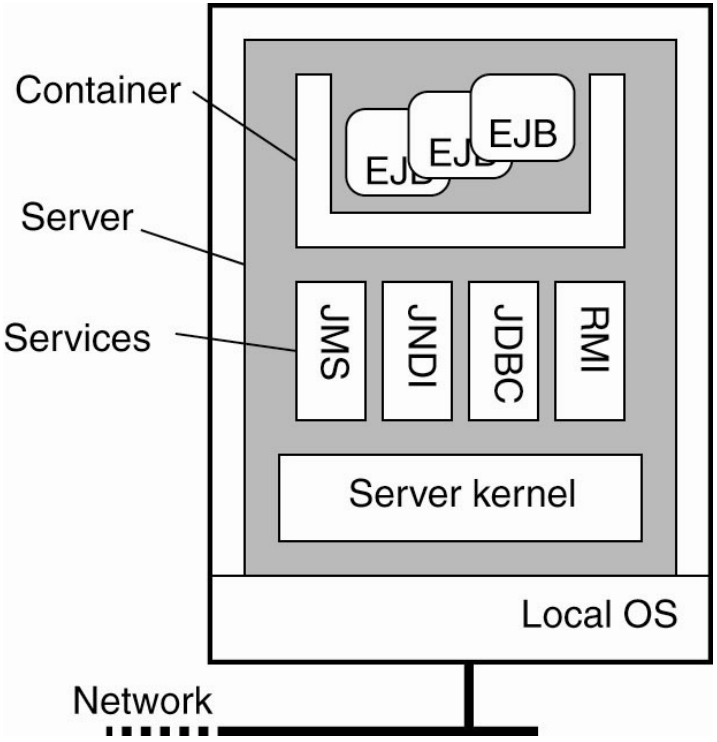
- Parameters,
  - pass local objects by value/result
  - and remote objects by reference



The organization to implement "*RMI (C, obj.method (L1, R1))*"

Run time system

- Distributed objects are supported by (services of) a run-time system
  - management: creation/destruction of objects, storage
  - communication: inspection of interfaces, binding, invocation
  - accessed via the container
- Java Enterprise Beans:
  - 4 types of objects
    - session beans
      - statefull or stateless
    - entity beans (persistent)
    - message-driven beans
  - JMS: *Java Message Service*
  - JNDI: *Java Naming and Directory Interface*
  - JDBC: *Java DataBase Connectivity*
  - RMI: *Remote Method Invocation*



Agenda

- Introduction
- Layered protocols and middleware
- Remote Invocation
  - (Remote) Procedure calls
  - (Remote) Method invocation
- **Message Oriented Middleware**
- Streaming

Message Oriented Middleware

- RPC & RMI are rather synchronous
  - synchronous in time: wait for reply
  - synchronous in space: shared data is known
  - functionality and communication coupled

- Look for communication models with better decoupling and other qualities:
  - message oriented: communication and functionality separated
    - more basic: plain point-to-point communication of data
    - more abstract: forms of indirect communication offering temporal and referential decoupling

Sockets

- **Sockets with TCP give a point-to-point byte oriented transport service**
- **The TCP service API**
  - **8 functions**
  - **2 roles**
    - **client**
    - **server**

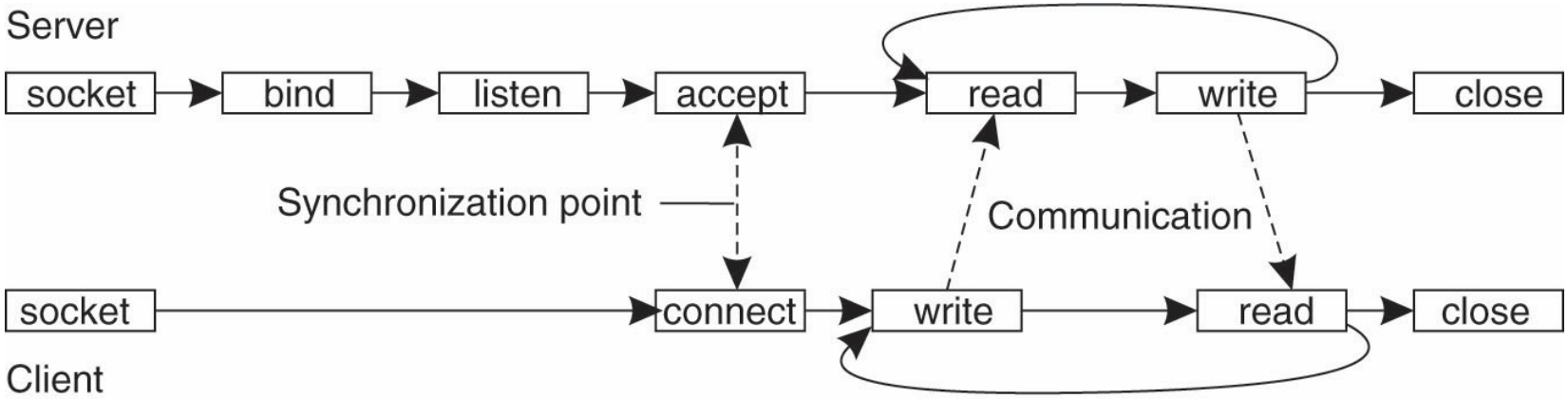
Operation	Description
socket	Create a new communication end point
bind	Attach a local address to a socket
listen	Tell operating system what the maximum number of pending connection requests should be
accept	Block caller until a connection request arrives
connect	Actively attempt to establish a connection
send	Send some data over the connection
receive	Receive some data over the connection
close	Release the connection

- **The quality of the service:**
  - **transient, discrete, connection oriented, buffered, reliable**
    - **reliability against packet loss, duplicates, and reordering, is traded for delay**
- **Rather basic interface**
  - **just send/receive**
  - ***basis for more advanced middleware services (see: protocol triangle)***
- **Note: there is also an unreliable datagram service: UDP**

Setup / communicate / destroy

- Client needs to know the transport-level address of the server
- End-to-end connection is characterized by 2 address pairs

  (ClientHost, ClientPort) ←→ (ServerHost, ServerPort)

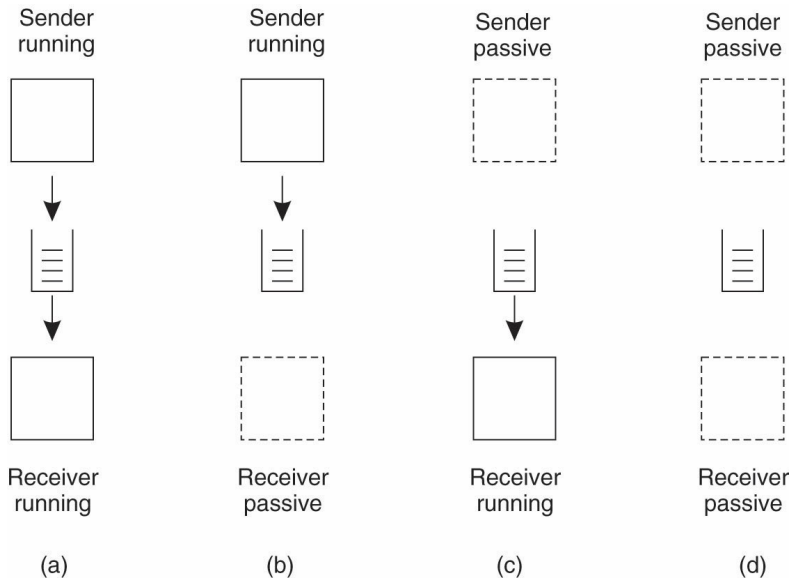


Connection setup using sockets

Message queuing systems

Typical behavior

- each application its own queue
- queues can only be read by their associated application
- applications communicate by sending messages to each other queues
- temporal-decoupling
  - 4 communication states



Message queuing systems qualities

- reliability
  - guaranteed eventual *delivery* no guarantee *when* and whether or not the message will be handled
- persistent, discrete, asynchronous
  - limited synchronization, e.g. upon handing off to the MQ system
  - perhaps *controllable* synchronization
  - persistence gives freedom to sender and receivers to go offline

Message queuing systems

- Message queuing systems
  - simple interface
    - note that the primitives use *queue* references, which makes them more abstract

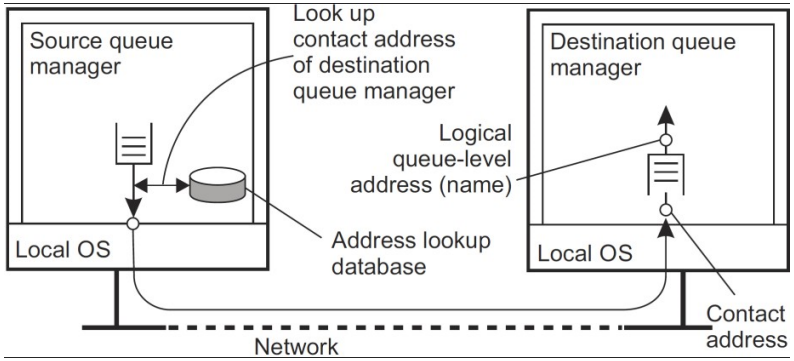
Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

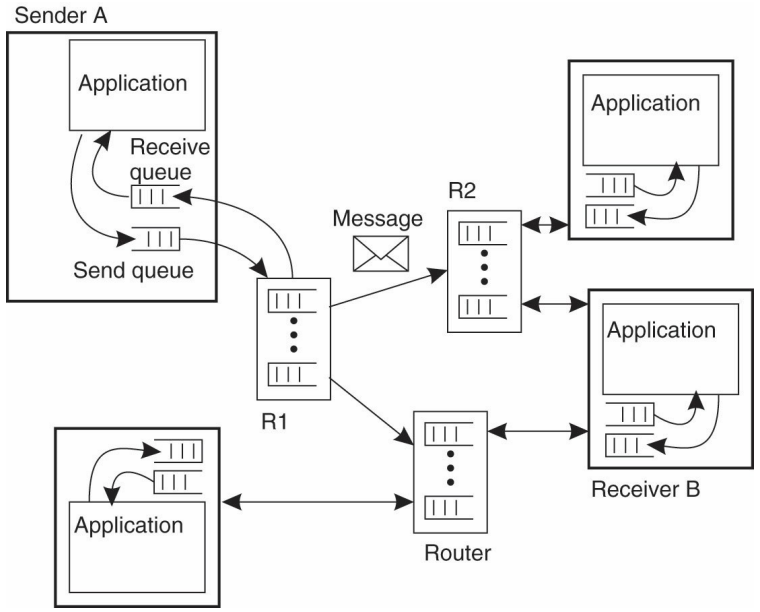
    - removal of first message implies FIFO delivery
    - in practical cases more management is possible (message and queue sizes, ordering, ...)
    - extension of the system with brokers allows usage of topics

Similar to email systems, but serving different purposes, not only a single service to end-users, but integrating services into new applications, and hence subject to different requirements.

General architecture of queuing systems

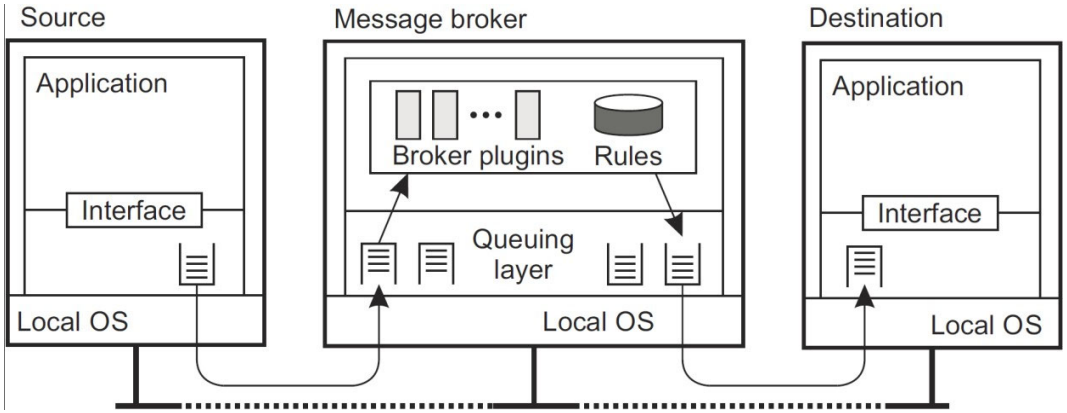
- Use (unique) *logical* names for message *queues*
- *Queue manager (QM)*
  - a process or a library
- The queuing layer forms an overlay (distributed set of QMs)
  - need to map queue names to transport addresses
- For scalability, the overlay is extended with routers (bottom)
  - scalability: why?
    - the complexity of maintaining a global overview
  - also allows for secondary processing (e.g. logging) and scalable multicasting



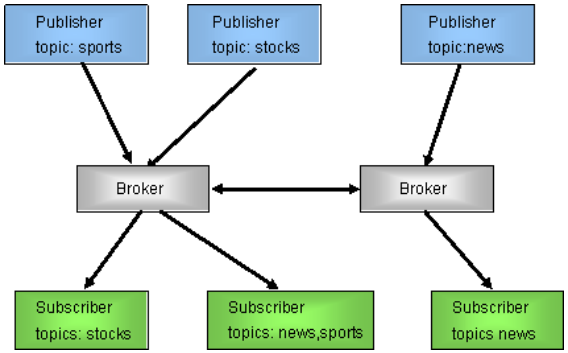


Message brokers

- Diverse applications
  - hence, diverse message formats
  - conversion required
    - perhaps taking application know-ledge into account
    - rules for conversion needed

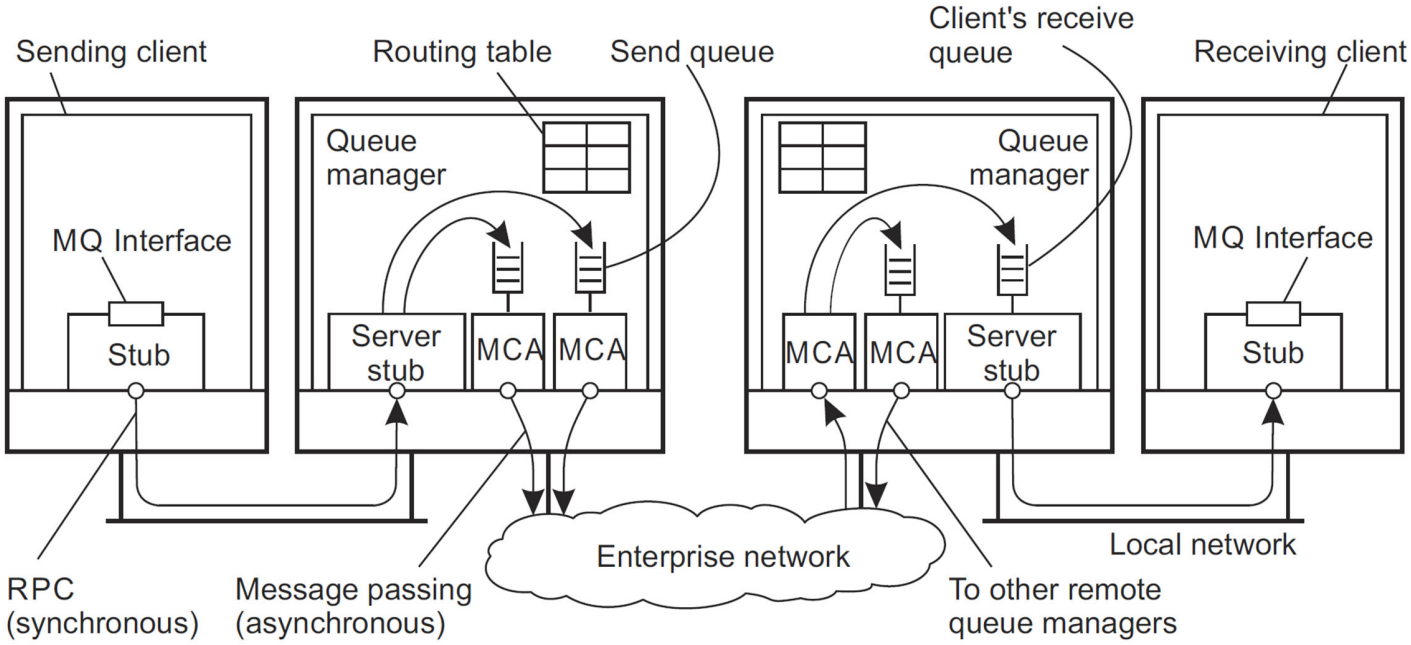


- Support for PubSub implementation
  - rules for filtering and forwarding needed
    - i.e matching publications to subscriptions



Examples of MQ-systems

- Websphere MQ (from IBM)



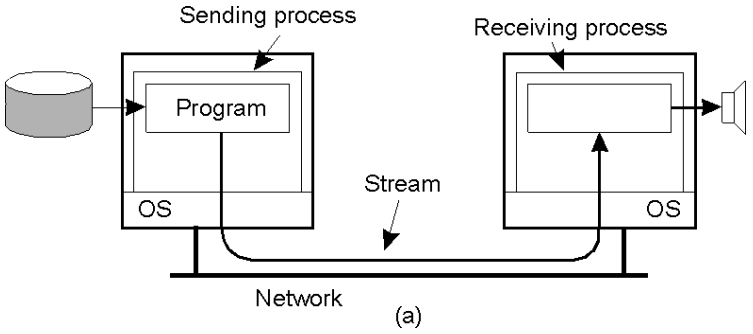
- JMS
- AMQP
  - More service oriented
  - RabbitMQ see https://www.rabbitmq.com/getstarted.html

Agenda

- Layered protocols and middleware
- (Remote) Procedure calls
- (Remote) Method invocation
- Message Oriented Middleware
- Streaming

Stream-Oriented Communication

- Media streaming
  - temporal relationships between data items play a crucial role
    - e.g. audio (mono, stereo), video, audio + video;
    - 20 μsec difference in channels distorts stereo;
    - 80 msec difference is acceptable for audio – video (asymmetric, lip-synchronization)
  - Synchronous transmission mode: bounded end-to-end delay (= bounded latency)
  - Isochronous transmission mode: bounded inter-packet delay (= bounded jitter)



- Soft (firm) real-time
  - missing a packet is a pity but no disaster
  - graceful reduction in quality

- General issue: *quality of service*
  - no discrete success/failure but range of qualities, decided dynamically
    - trade resources for quality
  - QoS guarantee is end-to-end concern, points along the route must cooperate
    - RSVP: resource reservation protocol

QoS-techniques (tactics)

- Buffering
  - smooths delay variation
- Priority routing
  - reduces latency and jitter
- Flow control (both route and rate)
- Resolution and quantization scaling
  - resolution: pixels per frame
  - quantization: bits per pixel
  - improves throughput
- Quality modes
  - multiple streams, e.g., least significant bits in separate stream
  - mode decides which streams are sent,
    - in general, a combination of QoS-levels per quality aspect