



POLITECNICO
MILANO 1863

Distributed Systems Communication

Gianpaolo Cugola

Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy

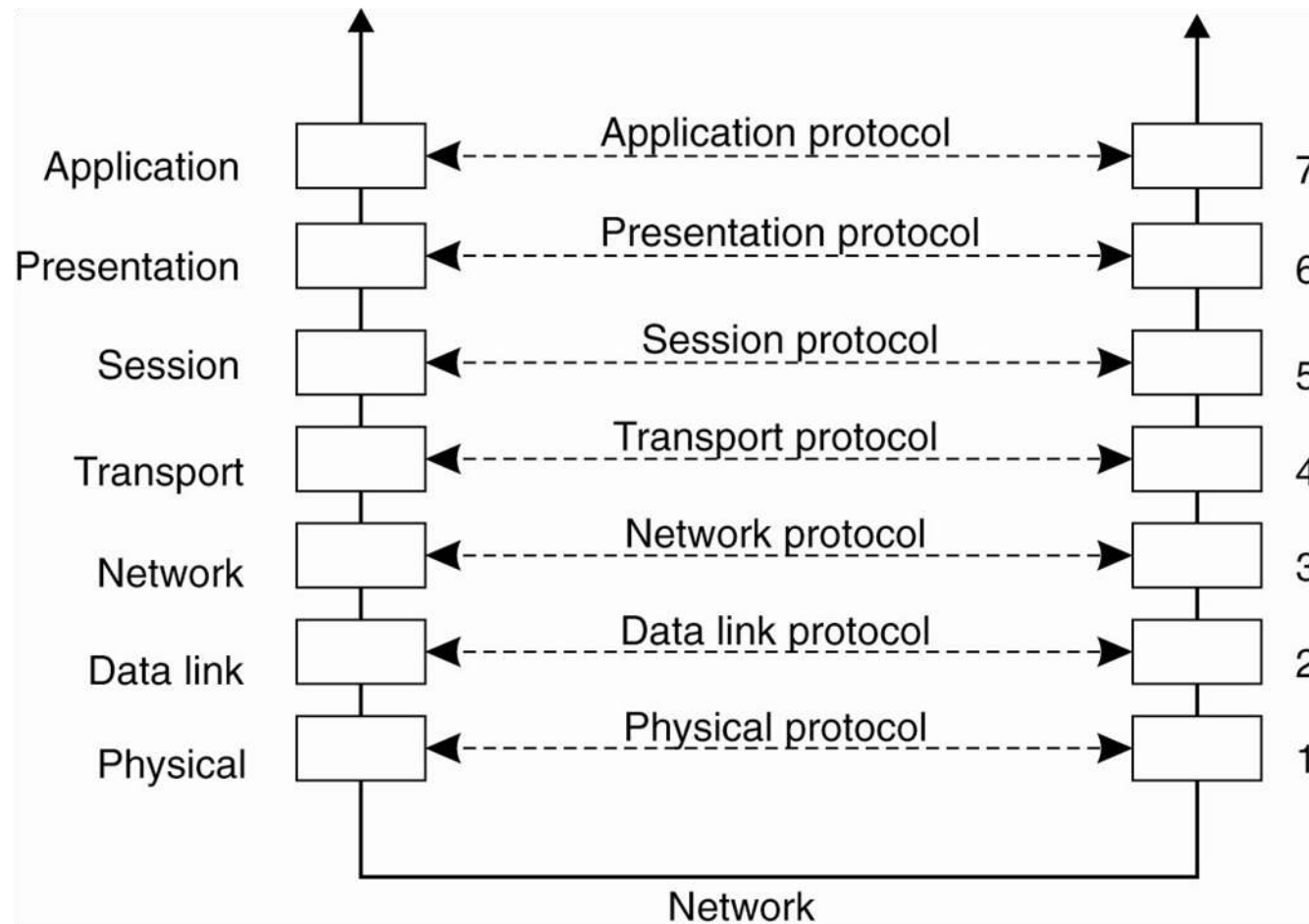
gianpaolo.cugola@polimi.it

<http://home.dei.polimi.it/cugola>

Contents

- **Fundamentals**
 - **Protocols and protocol stacks**
 - **Middleware**
- Remote procedure call
 - Fundamentals
 - Discovering and binding
 - Sun and DCE implementations
- Remote method invocation
 - Fundamentals
- Message oriented communication
 - Fundamentals
 - Message passing (sockets and MPI)
 - Message queuing
 - Publish/subscribe
- Stream-oriented communication
 - Fundamentals
 - Guaranteeing QOS

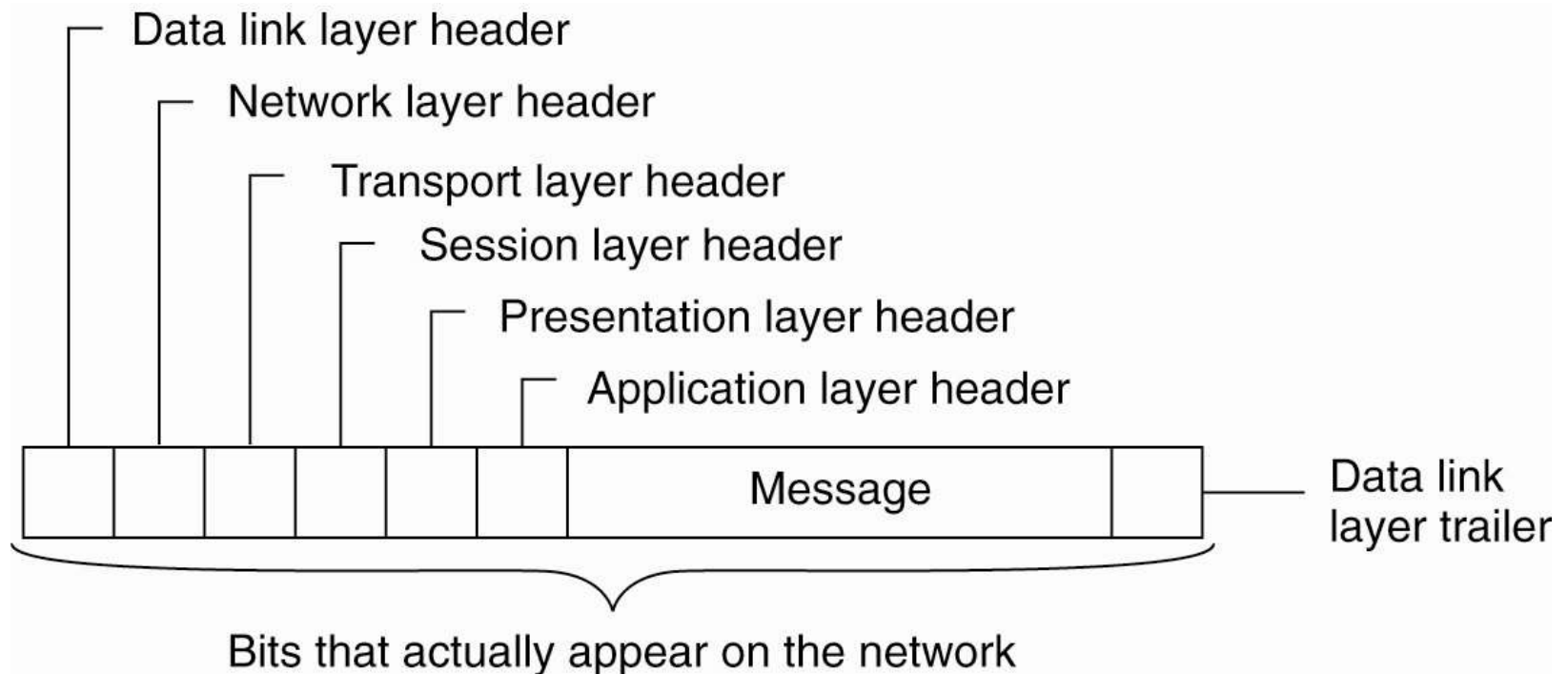
Layered protocols: The OSI model



The OSI model: Recap

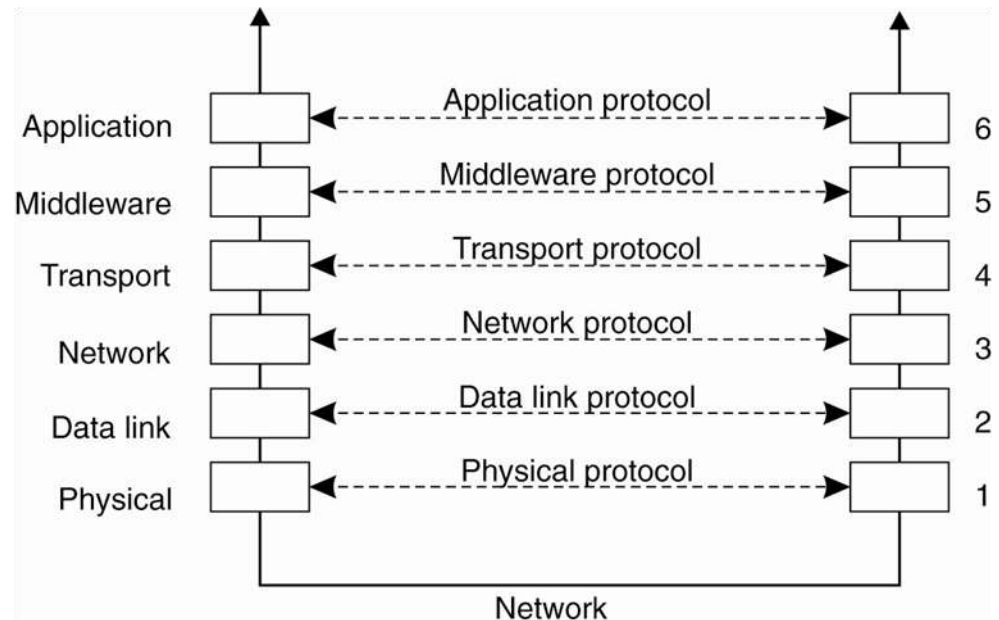
- Low level layers
 - Physical layer: describes how bits are transmitted between two directly connected nodes
 - Data link layer: describes how a series of bits is packed into a frame to allow for error and flow control
 - Network layer: describes how packets in a network of computers are to be routed
- Transport layer
 - Describes how data is transmitted among two nodes, offering a service independent from the lower layers
 - It provides the actual communication facilities for most distributed systems
 - Standard Internet protocols
 - TCP: connection-oriented, reliable, stream-oriented communication
 - UDP: unreliable (best-effort) datagram communication
- Higher level layers
 - Merged together in the current, Internet practice

Layered Protocols: Encapsulation



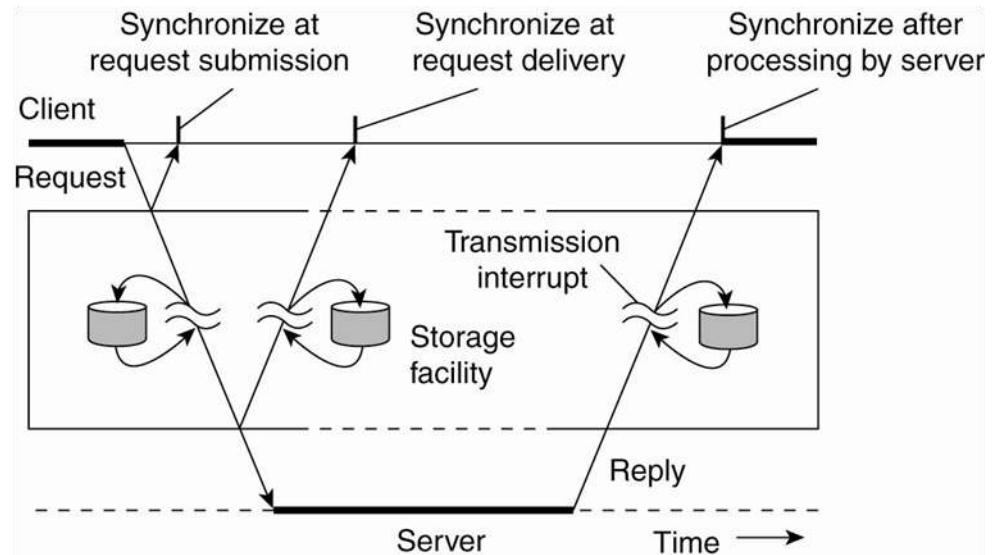
Middleware as a protocol layer

- Middleware includes common services and protocols that can be used by many different applications
 - (Un)marshaling of data, necessary for integrated systems
 - Naming protocols, to allow easy sharing of resources
 - Security protocols for secure communication
 - Scaling mechanisms, such as for replication and caching
- What remains are truly application-specific protocols...



Types of Communication

- Middleware may offer different form of communication:
 - Transient vs. persistent
 - Synchronous vs. asynchronous (various forms)
- Popular combinations
 - Transient communication with synchronization after processing
 - Persistent communication with synchronization at request submission

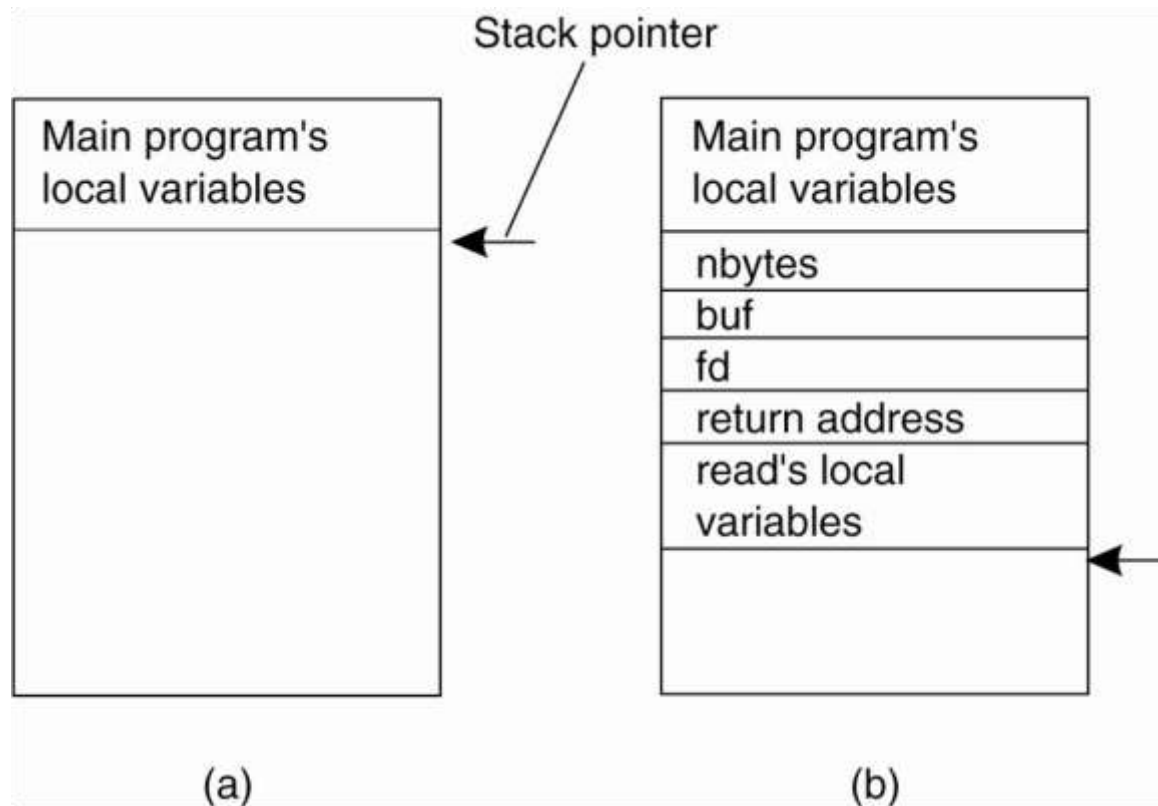


Contents

- Fundamentals
 - Protocols and protocol stacks
 - Middleware
- **Remote procedure call**
 - **Fundamentals**
 - **Discovering and binding**
 - **Sun and DCE implementations**
- Remote method invocation
 - Fundamentals
- Message oriented communication
 - Fundamentals
 - Message passing (sockets and MPI)
 - Message queuing
 - Publish/subscribe
- Stream-oriented communication
 - Fundamentals
 - Guaranteeing QOS

Local procedure call

- Parameter passing in a local procedure call
 - the stack before (a) and after (b) the call to:
`count = read(fd, buf, nbytes)`

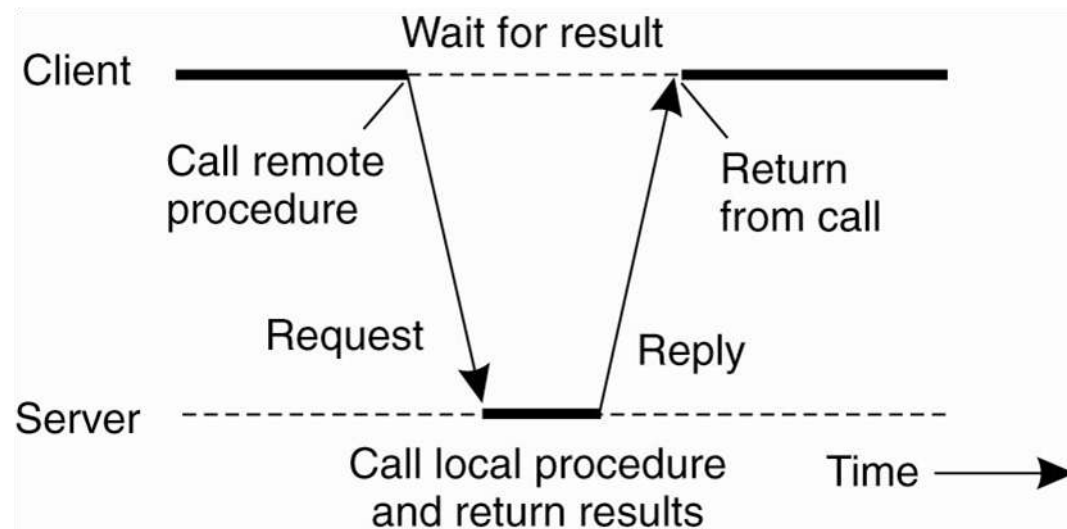


Passing parameters to a procedure

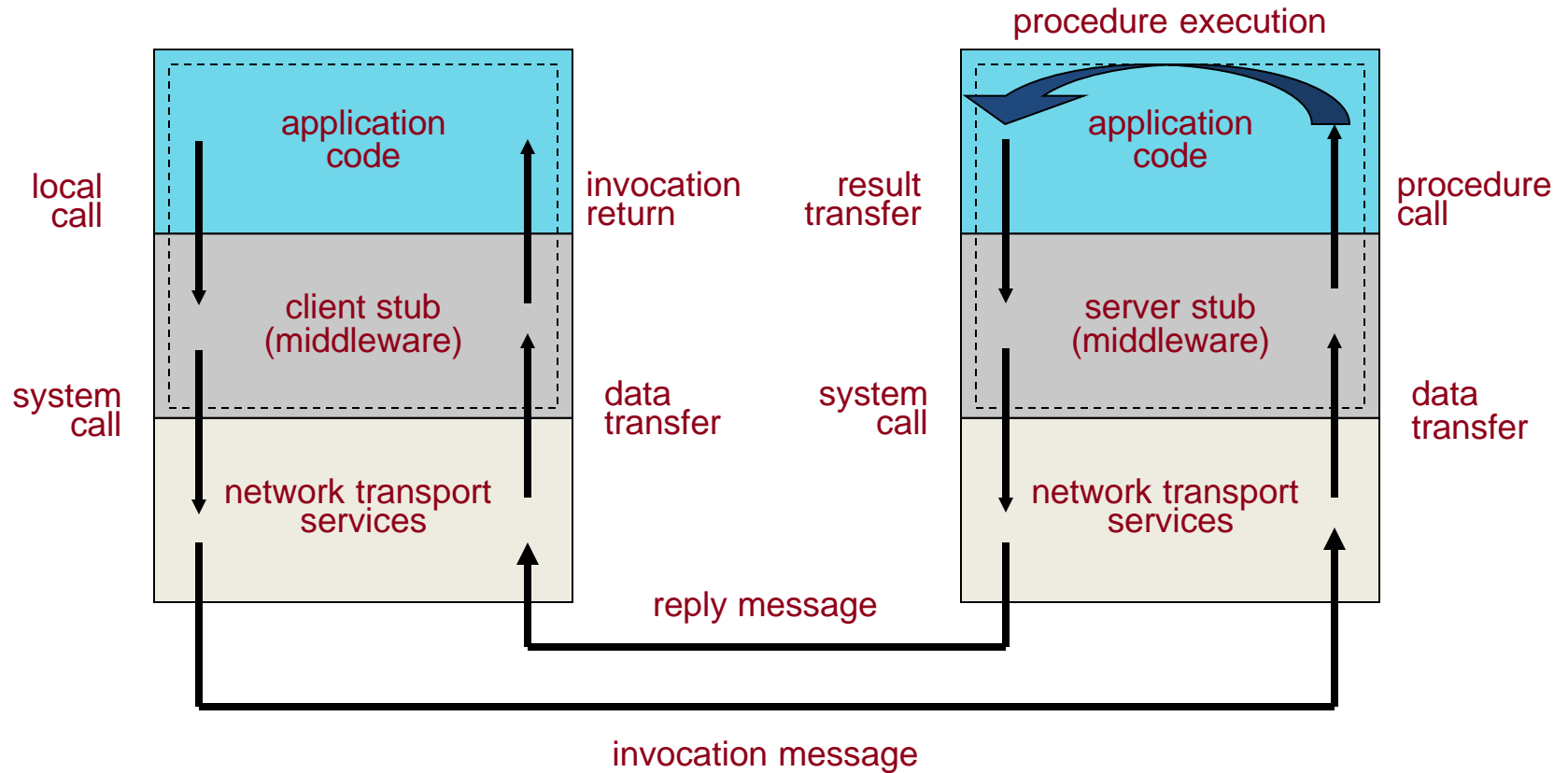
- Different mechanisms to pass parameters
 - *By value*. Like in C when passing basic data types
 - *By reference*. Like in C when passing arrays or in Java when passing objects
 - *By copy/restore*. Similar but slightly different than previous one

From local to remote procedure call

- Considerations
 - Application developers are familiar with procedure call
 - Well-engineered procedures operate in isolation (black box) helping structuring code
 - There is no fundamental reason not to execute procedures on separate machine
- Conclusion
 - Remote communication can be hidden by using procedure-call mechanism

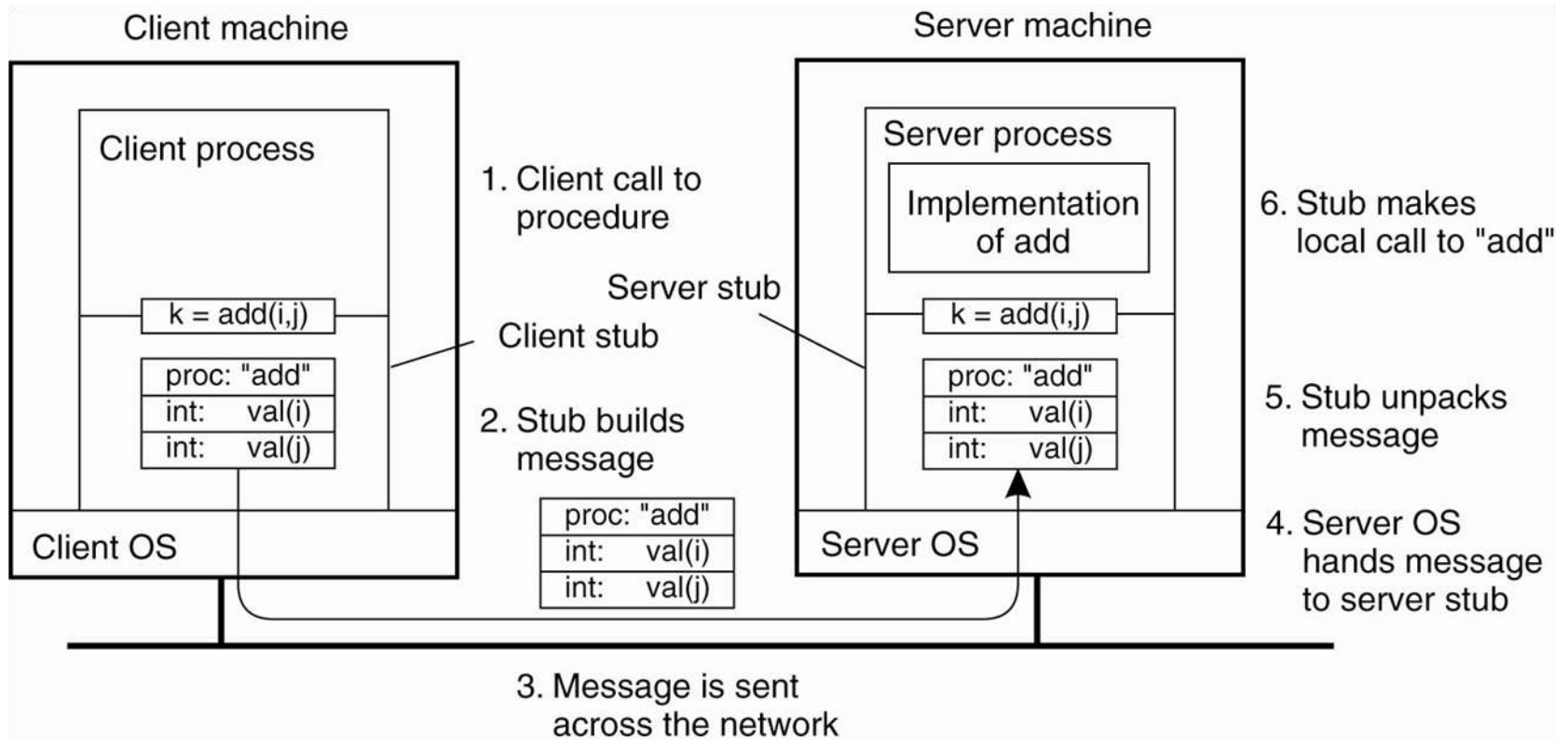


RPC: How it works



RPC in detail

(when passing parameters by value)



Parameter passing: Marshalling and serialization

- Passing a parameter poses two problems:
 - Structured data (e.g., structs/records, objects) must be ultimately flattened in a byte stream
 - Called *serialization* (or pickling, in the context of OODBMSs)
 - Hosts may use different data representations (e.g., little endian vs. big endian, EBCDIC vs. ASCII) and proper conversions are needed
 - Called *marshalling*
- Middleware provides automated support:
 - The marshalling and serialization code is automatically generated from and becomes part of the stubs
 - Enabled by:
 - A language/platform independent representation of the procedure's signature, written using an *Interface Definition Language* (IDL)
 - A data representation format to be used during communication

The role of IDL

- The *Interface Definition Language* (IDL) raises the level of abstraction of the service definition
 - It separates the service *interface* from its *implementation*
 - The language comes with “mappings” onto target languages (e.g., C, Pascal, Python...)
- Advantages:
 - Enables the definition of services in a language-independent fashion
 - Being defined formally, an IDL description can be used to automatically generate the service interface code in the target language

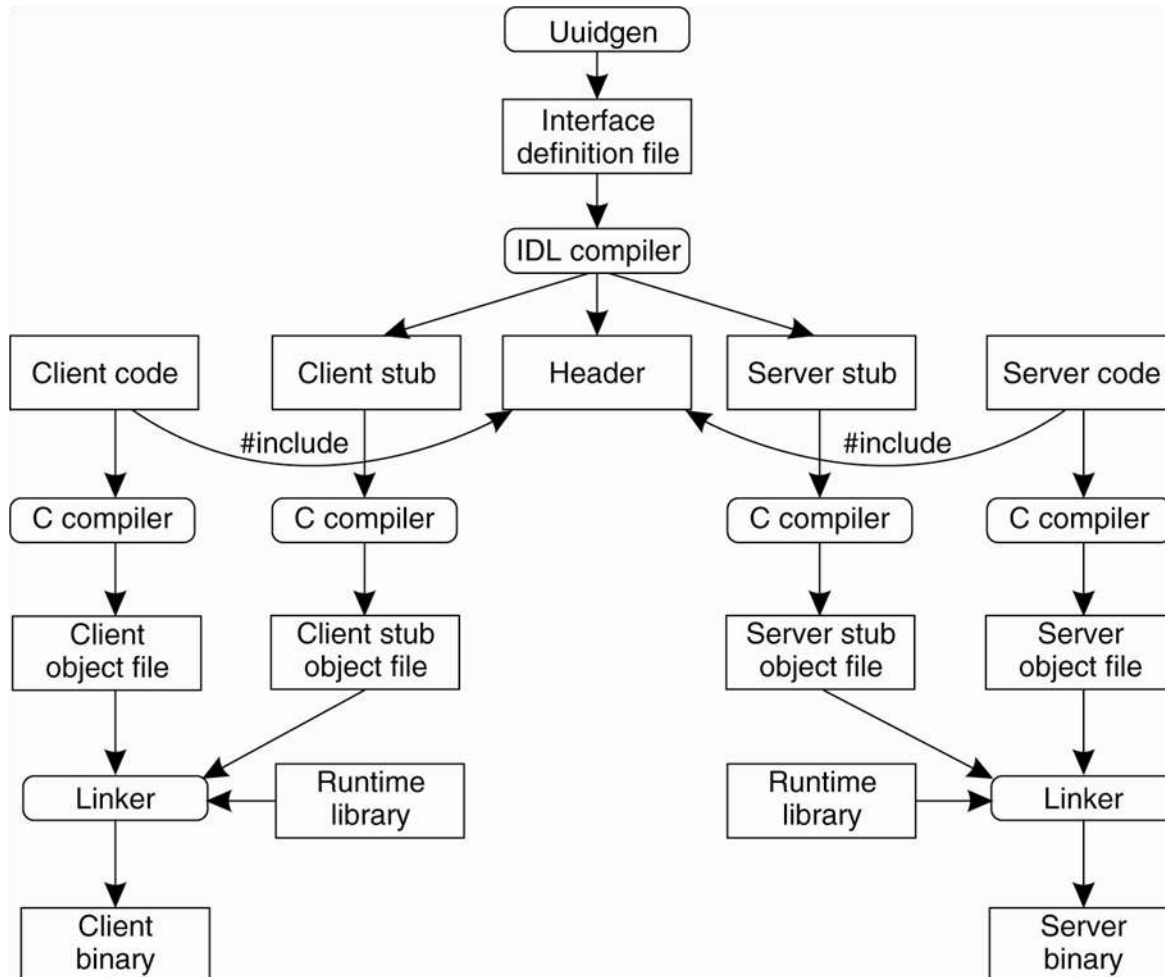
Passing parameters by reference

- How to pass a parameter by reference?
 - Many languages do not provide a notion of reference, but only of pointer
 - A pointer is meaningful only within the address space of a process...
- Often, this feature is simply not supported (as in Sun's solution)
- Otherwise, a possibility is to use call by value/result instead of call by reference
 - Semantics is different!
 - Works with arrays but not with arbitrary data structures
 - Optimizations are possible if input- or output-only

RPC in practice

- Sun Microsystems' RPC (also called Open Network Computing RPC) is the *de facto* standard over the Internet
 - At the core of NFS, and many other (Unix) services
 - Data format specified by XDR (eXternal Data Representation)
 - Transport can use either TCP or UDP
 - Parameter passing:
 - Only pass by copy is allowed (no pointers). Only one input and one output parameter
 - Provision for DES security
- The Distributed Computing Environment (DCE) is a set of specifications and a reference implementation
 - From the Open Group, no-profit standardization organization
 - Several invocation semantics are offered
 - At most once, idempotent, broadcast
 - Several services are provided on top of RPC:
 - Directory service, distributed time service, distributed file service
 - Security is provided through Kerberos
 - Microsoft's DCOM and .Net remoting are based on DCE

RPC in practice



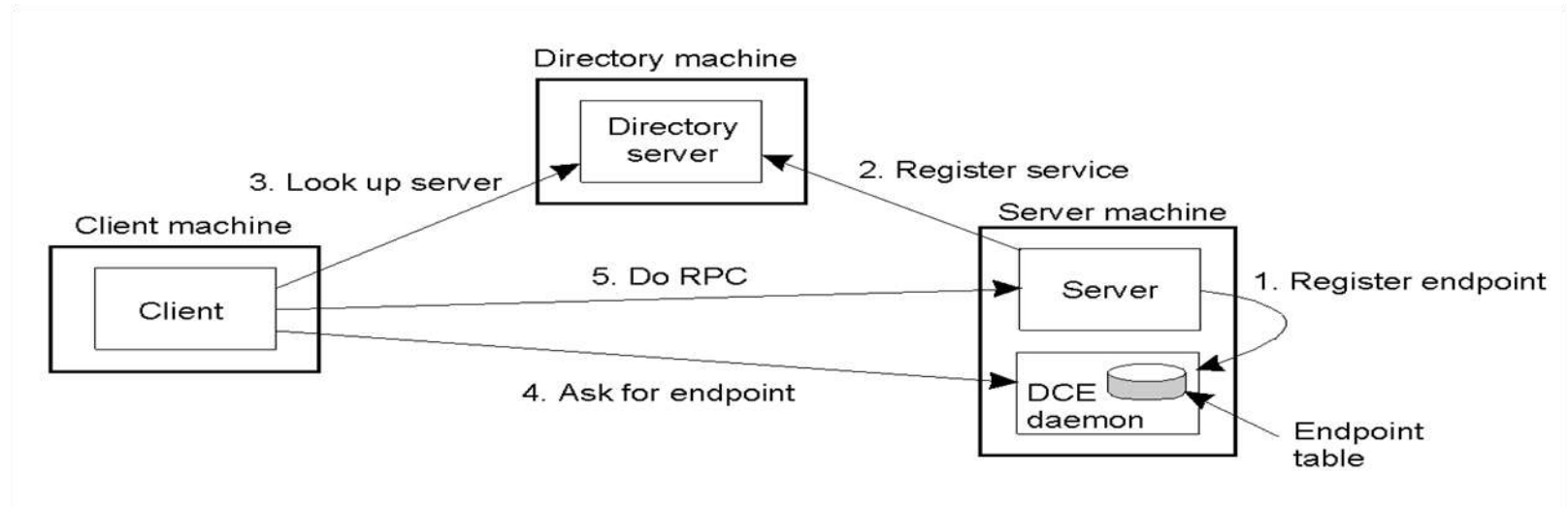
Binding the client to the server

- Problem: find out which server (process) provides a given service
 - Hard-wiring this information in the client code is highly undesirable
 - Two distinct problems:
 - Find out where the server process is
 - Find out how to establish communication with it

Sun's solution

- Introduce a daemon process (`portmap`) that binds calls and server/ports:
 - The server picks an available port and tells it to `portmap`, along with the service identifier
 - Clients contact a given `portmap` and:
 - Request the port necessary to establish communication
- `portmap` provides its services only to local clients, i.e., it solves only the second problem
 - The client must know in advance where the service resides
 - However:
 - A client can multicast a query to multiple daemons
 - More sophisticated mechanisms can be built or integrated
 - e.g., directory services

DCE's solution



- The DCE daemon works like portmap
- The directory server (aka binder daemon) enables location transparency:
 - Client need not know in advance where the service is: they only need to know where the directory service is
 - In DCE, the directory service can actually be distributed
 - To improve scalability over many servers
 - Step 3 is needed only once per session

Dynamic activation

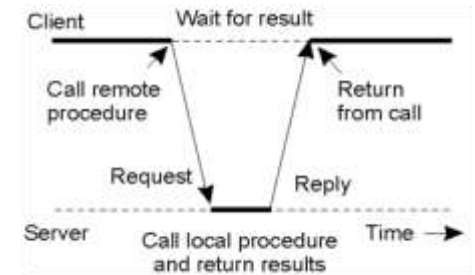
- Problem: server processes may remain active even in absence of requests, wasting resources
- Solution: introduce another (local) server daemon that:
 - Forks the process to serve the request
 - Redirects the request if the process is already active
 - Clearly, the first request is served less efficiently
- In Sun RPC:
 - inetd daemon
 - The mapping between requested service and server process is stored in a configuration file (/etc/services)

Lightweight RPC

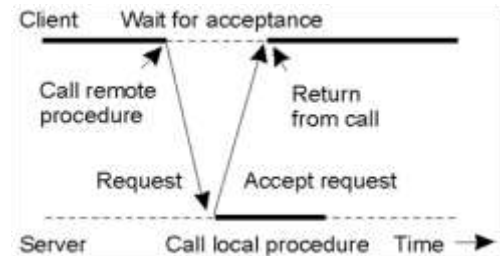
- It is natural to use the same primitives for inter-process communication, regardless of distribution
 - But using conventional RPC would lead to wasted resources: no need for TCP/UDP on a single machine!
- Lightweight RPC: message passing using local facilities
 - Communication exploits a private shared memory region
 - Lightweight RPC invocation:
 - Client stub copies the parameters on the shared stack and then performs a system call
 - Kernel does a context switch, to execute the procedure in the server
 - Results are copied on the stack and another system call + context switch brings execution back to the client
 - Advantages:
 - Uses less threads/processes (no need to listen on a channel)
 - 1 parameter copy instead of 4 (2 x (stub→kernel + kernel→stub))
- Similar concepts used in practice in DCOM and .NET

Asynchronous RPC

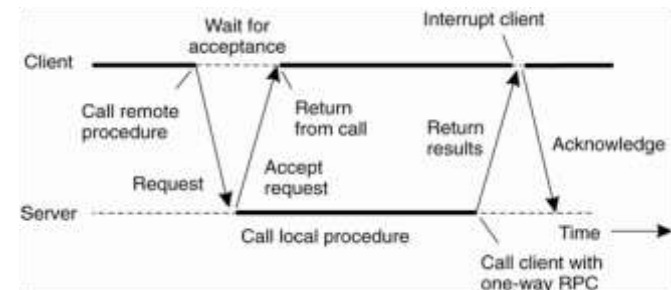
- RPC preserves the usual call behavior
 - The caller is suspended until the callee is done
- Potentially wastes client resources
 - Evident if no return value is expected
 - In general, concurrency could be increased
- Many variants of asynchronous RPC (with different semantics):
 - If no result is needed execution can resume after an acknowledgment is received from the server
 - One-way RPC returns immediately
 - May cause reliability issues: “Maybe semantics”
 - To deal with results, the callee may (asynchronously) invoke the caller back
 - Or invocation may return immediately a *promise* (or *future*), later polled by the client to obtain the result



synchronous



**asynchronous
(one possible)**



deferred synchronous

Batched vs. queued RPC

- Sun RPC includes the ability to perform *batched* RPC
 - RPCs that do not require a result are buffered on the client
 - They are sent all together when a non-batched call is requested (or when a timeout expires)
 - Enables yet another form of asynchronous RPC
- A similar concept can be used to deal with mobility (as in the Rover toolkit by MIT):
 - If a mobile host is disconnected between sending the request and receiving the reply, the server periodically tries to contact the mobile host and deliver the reply
 - Requests and replies can come through different channels
 - Depending on network conditions and application requirements, the network scheduler module may decide to:
 - Send requests in batches
 - Compress the data
 - Reorder requests and replies in a non-FIFO order, e.g., to suit application-specified priorities
 - Promises are used at the client

Contents

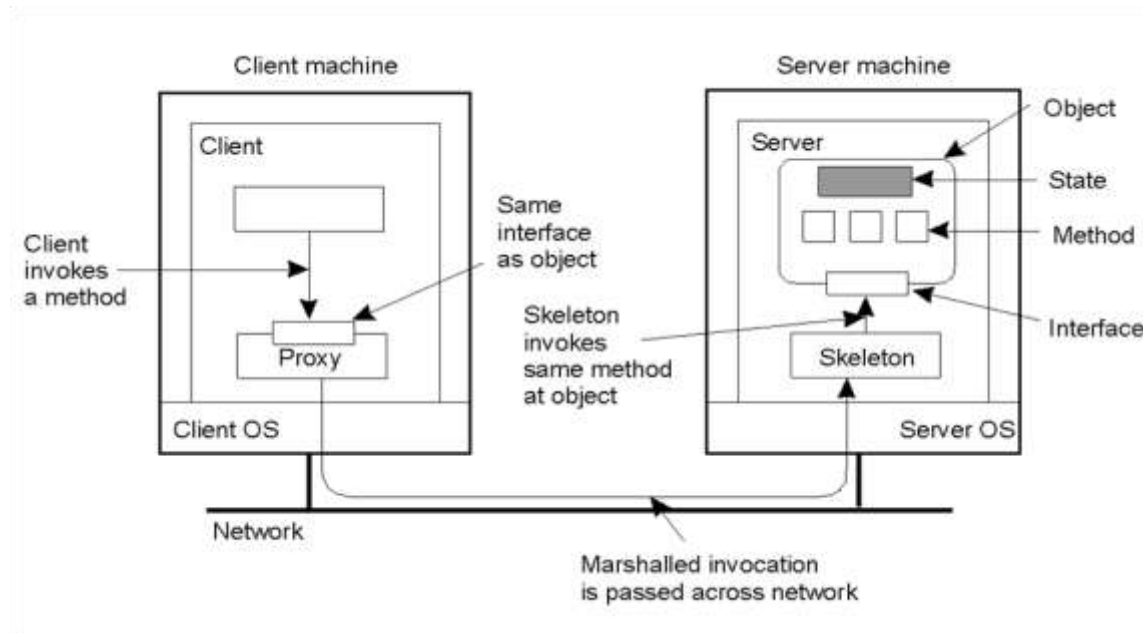
- Fundamentals
 - Protocols and protocol stacks
 - Middleware
- Remote procedure call
 - Fundamentals
 - Discovering and binding
 - Sun and DCE implementations
- **Remote method invocation**
 - **Fundamentals**
- Message oriented communication
 - Fundamentals
 - Message passing (sockets and MPI)
 - Message queuing
 - Publish/subscribe
- Stream-oriented communication
 - Fundamentals
 - Guaranteeing QOS

Remote method invocation

- Same idea as RPC, different programming constructs
 - The aim is to obtain the advantages of OOP also in the distributed setting
- Important difference: remote object references can be passed around
 - Need to maintain the aliasing relationship
- Shares many of the core concepts and mechanisms with RPC
 - Sometimes built on top of an RPC layer

Interface definition language

- In RPC, the IDL separates the interface from the implementation
 - To handle platform/language heterogeneity
- Such separation is one of the basic OO principles
 - It becomes natural to place the object interface on one host, and the implementation on another
- The IDLs for distributed objects are much richer
 - Inheritance, exception handling, ...



Remote method invocation in practice

- Java RMI
 - Single language/platform (Java and the Java Virtual Machine)
 - Easily supports passing parameters by reference or “by value” even in case of complex objects
 - Supports for downloading code (code on demand)
- OMG CORBA
 - Multilanguage/multiplatform
 - Supports passing parameters by reference or by value
 - If objects are passed by value (valuetype) it is up to the programmer to guarantee the same semantics for methods on the sender and receiver sides

Contents

- Fundamentals
 - Protocols and protocol stacks
 - Middleware
- Remote procedure call
 - Fundamentals
 - Discovering and binding
 - Sun and DCE implementations
- Remote method invocation
 - Fundamentals
- **Message oriented communication**
 - **Fundamentals**
 - **Message passing (sockets and MPI)**
 - **Message queuing**
 - **Publish/subscribe**
- Stream-oriented communication
 - Fundamentals
 - Guaranteeing QOS

Message oriented communication

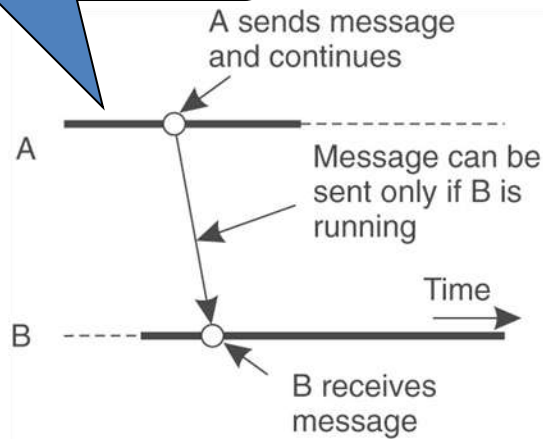
- RPC/RMI foster a synchronous model
 - Natural programming abstraction, but:
 - Supports only point-to-point interaction
 - Synchronous communication is expensive
 - Intrinsically tight coupling between caller and callee, leads to “rigid” architectures
- Message oriented communication:
 - Centered around the (simpler) notion of one-way message/event
 - Usually asynchronous (several forms)
 - Often supporting persistent communication
 - Often supporting multi-point interaction
 - Brings more decoupling among components

Types of communication

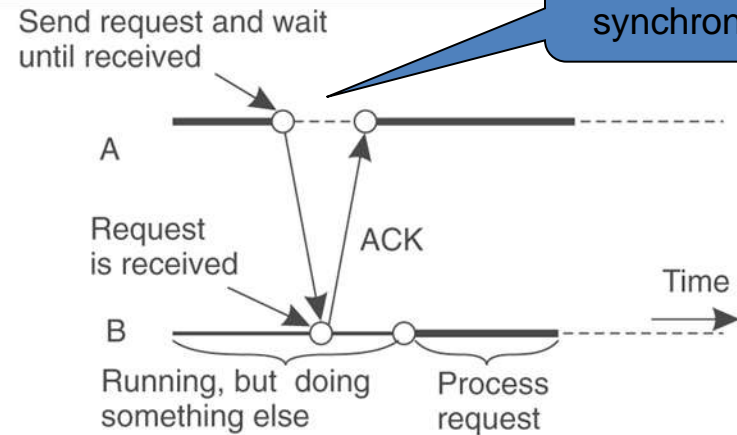
- Synchronous vs. asynchronous
 - Synchronous: the sender is blocked until the recipient has stored (or received, or processed) the message
 - Asynchronous: the sender continues immediately after sending the message
- Transient vs. persistent
 - Transient: sender and receiver must both be running for the message to be delivered
 - Persistent: the message is stored in the communication system until it can be delivered
- Several alternatives (and combinations) are provided in practice

Transient communication

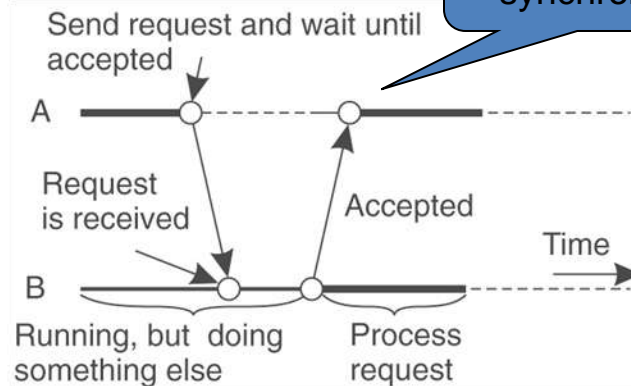
Pure Asynchronous



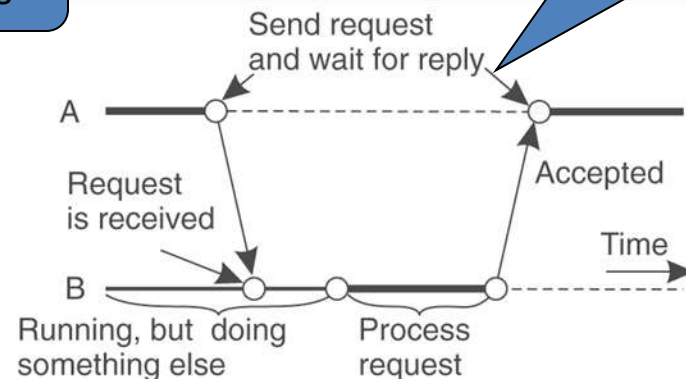
Receipt-based synchronous



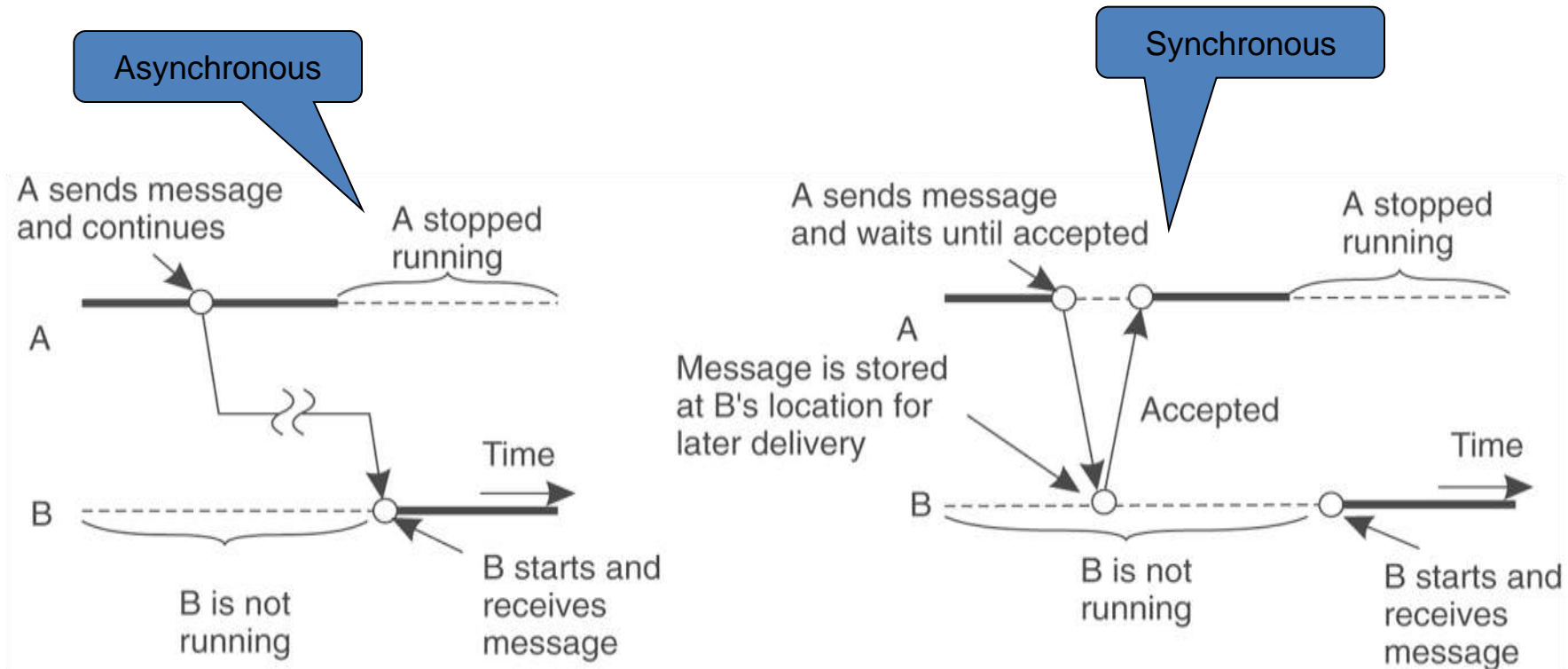
Delivery-based synchronous



Response-based synchronous

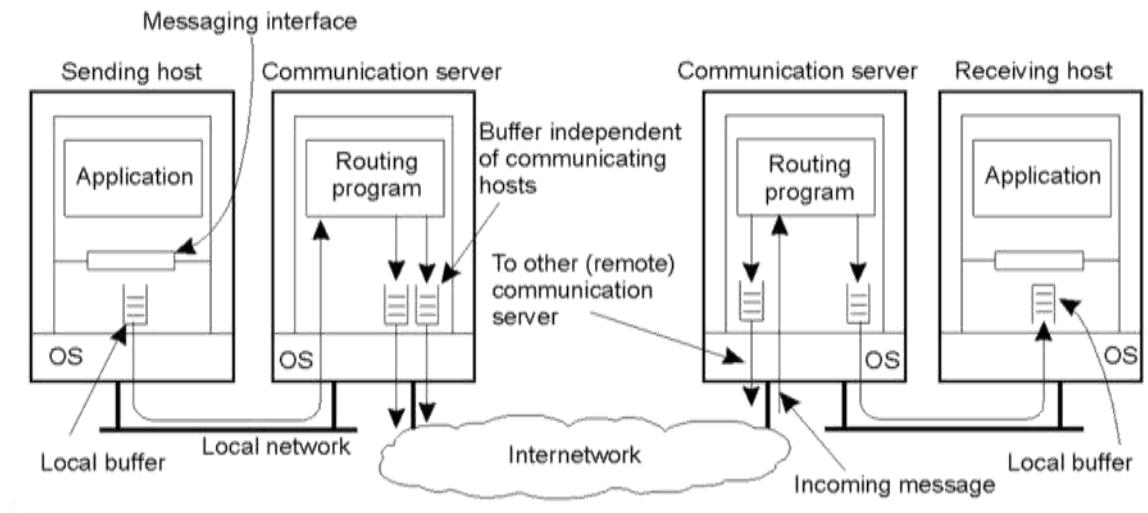


Persistent communication



Reference model

- The most straightforward form of message oriented communication is *message passing*
 - Typically directly mapped on/provided by the underlying network OS functionality (e.g., socket)
 - A (kind of) middleware provides another form of message passing called MPI
- *Message queuing* and *publish/subscribe* are two different models provided at the middleware layer
 - By several “communication servers”
 - Through what is nowadays called an *overlay network*

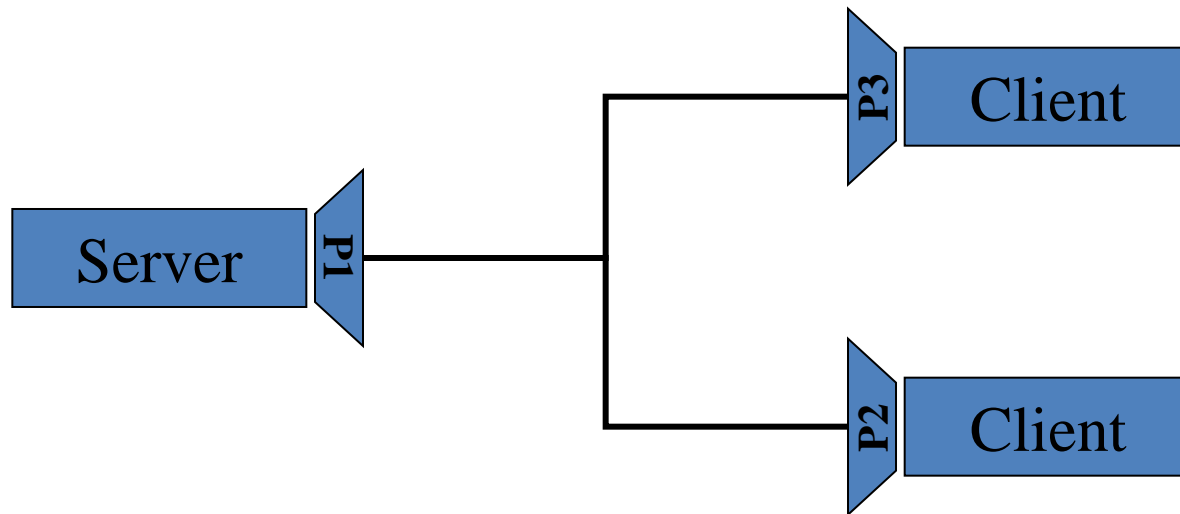


From network protocols to communication services

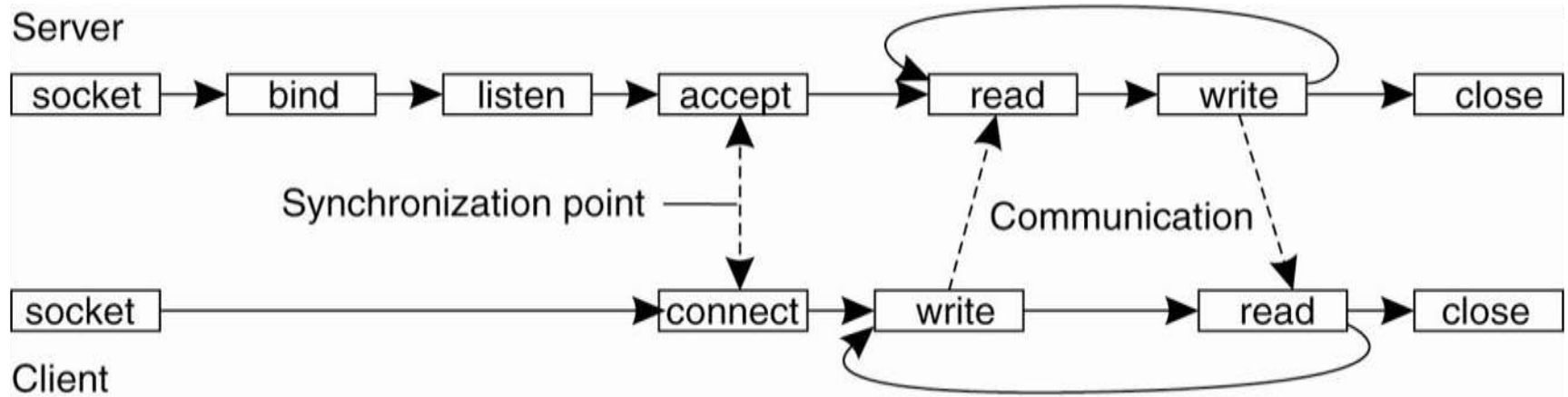
- Unicast TCP and UDP (and multicast IP) are well known network protocols
 - The related RFCs describe how they work in practice (on top of IP)
 - But how a poor programmer can take advantage of such protocols?
- Berkeley sockets are the answer!
 - First appeared in Unix BSD in 1982
 - Today available for every platform
- Sockets provide a common abstraction for inter-process communication
 - Unix and Internet sockets exist. Here we are interested in the latter
 - Allows for connection-oriented (stream i.e., TCP) or connectionless (datagram, i.e., UDP) communication

Stream sockets: Fundamentals

- The server accepts connection on a port
- The client connects to the server
- Each (connected) socket is uniquely identified by 4 numbers: The IP address of the server, its “incoming” port, the IP address of the client, its “outgoing” port

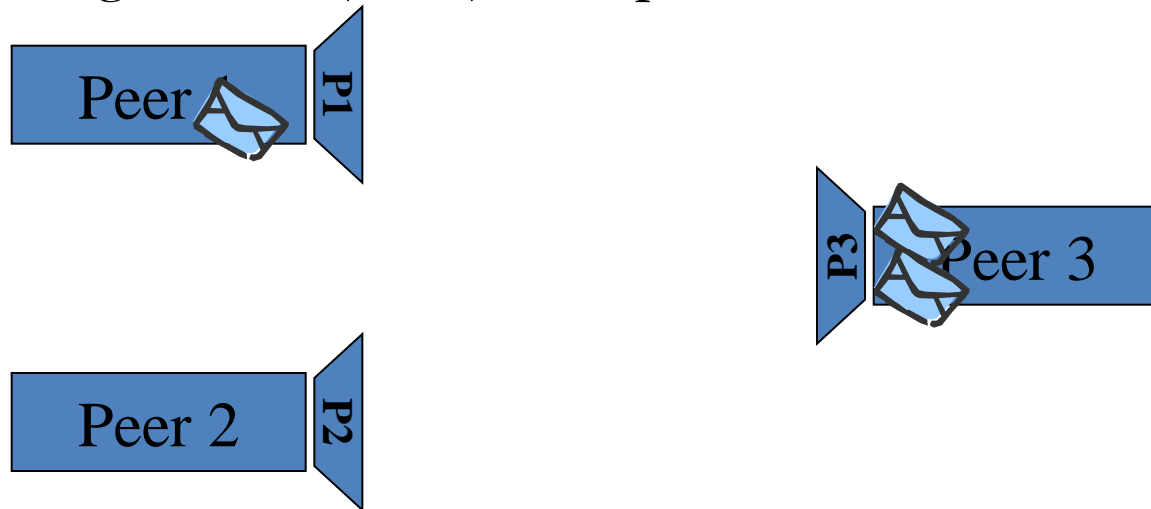


Stream sockets in C

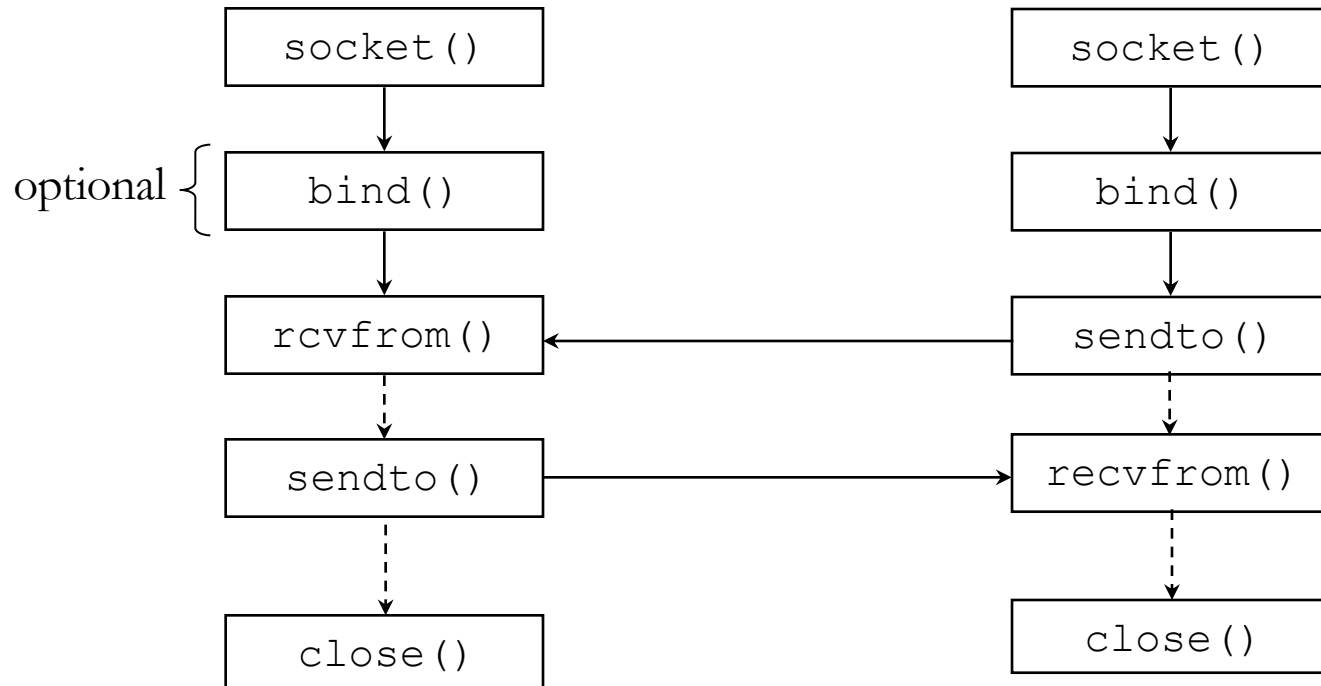


Datagram sockets: Fundamentals

- Client and server use the same approach to send and receive datagrams
- Both create a socket bound to a port and use it to send and receive datagrams
- There is no connection and the same socket can be used to send (receive) datagrams to (from) multiple hosts



Datagram sockets in C



Multicast sockets

- IP multicast is a network protocol to efficiently deliver UDP datagrams to multiple recipients
 - The Internet Protocol reserve a class D address space, from 224.0.0.0 to 239.255.255.255, to multicast *groups*
- The socket API for multicast communication is similar to that for datagram communication
 - Component interested in receiving multicast datagrams addressed to a specific group must *join* the group (using the `setsockopt` call)
 - Groups are open: It is not necessary to be a member of a group in order to send datagrams to the group
 - As usual it is also necessary to specify a *port*
 - It is used *by the OS* to decide which process on the local machine to route packets to
- Note: most routers are configured to *not* route multicast packets outside the LAN

MPI: Fundamentals

- Limitation of sockets
 - Low level
 - Protocol independent (and so awkward to use)
- In high performance networks (e.g., clusters of computers) we need higher level primitives for asynchronous, transient communication...
- ...providing different services besides pure read and write
- MPI was the (platform independent) answer

MPI: The model and main API

- Communication takes place within a known group of processes
- Each process within a group is assigned a local id
 - The pair (groupID, processID) represents a source or destination address
 - Messages can also be sent in broadcast to the entire group (MPI_Bcast, MPI_Reduce, MPI_Scatter, MPI_Gather)
- No support for fault tolerance (crashes are supposed to be fatal)
- Main MPI primitives

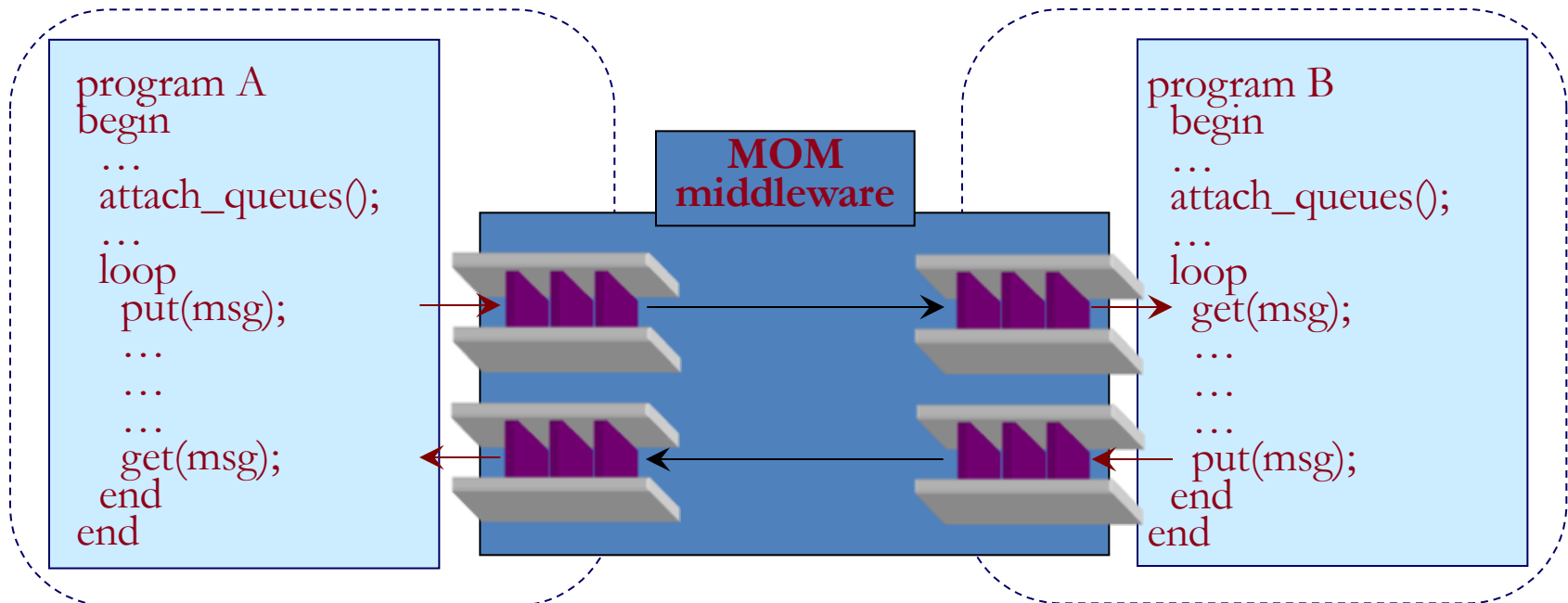
Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_irecv	Check if there is an incoming message, but do not block
MPI_recv	Receive a message; block if there is none
MPI_isend	Pass reference to outgoing message, and continue

Message queuing

- Point-to-point persistent asynchronous communication
 - Typically guarantee only eventual insertion of the message in the recipient queue (no guarantee about the recipient's behavior)
 - Communication is decoupled in time and space
 - Can be regarded as a generalization of the e-mail
- Intrinsically peer-to-peer architecture
- Each component holds an input queue and an output queue
- Many commercial systems:
 - IBM MQSeries (now WebSphere MQ), DECmessageQ, Microsoft Message Queues (MSMQ), Tivoli, Java Message Service (JMS), ...

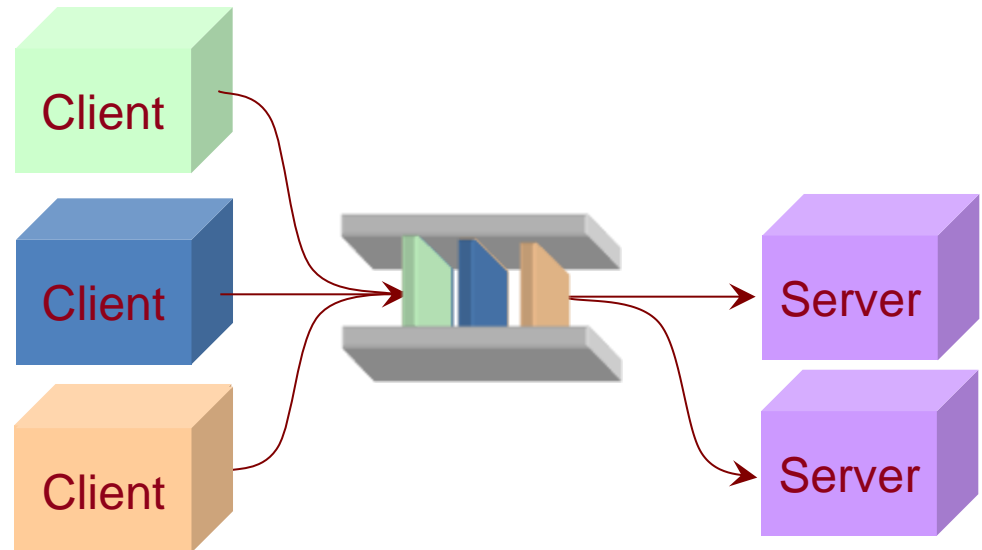
Queuing: Communication primitives

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



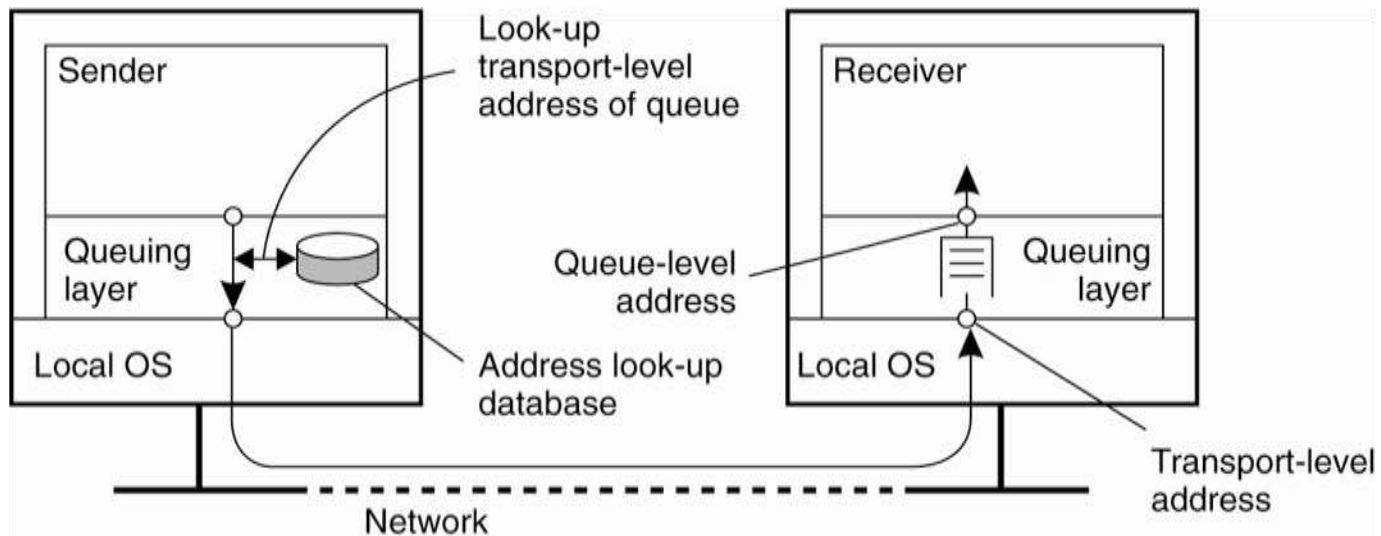
Client-Server with queues

- Clients send requests to the server's queue
- The server asynchronously fetches requests, processes them, and returns results in the clients' queues
 - Thanks to persistency and asynchronicity, clients need not remain connected
 - Queue sharing simplifies load balancing



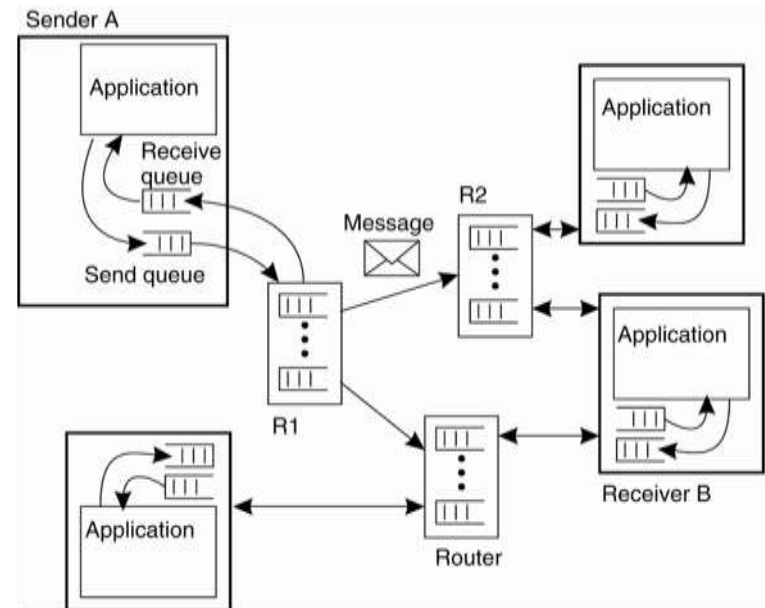
Message queuing: Architectural Issues

- Queues are identified by symbolic names
 - Need for a lookup service, possibly distributed, to convert queue-level addresses in network addresses
 - Often pre-deployed static topology/naming



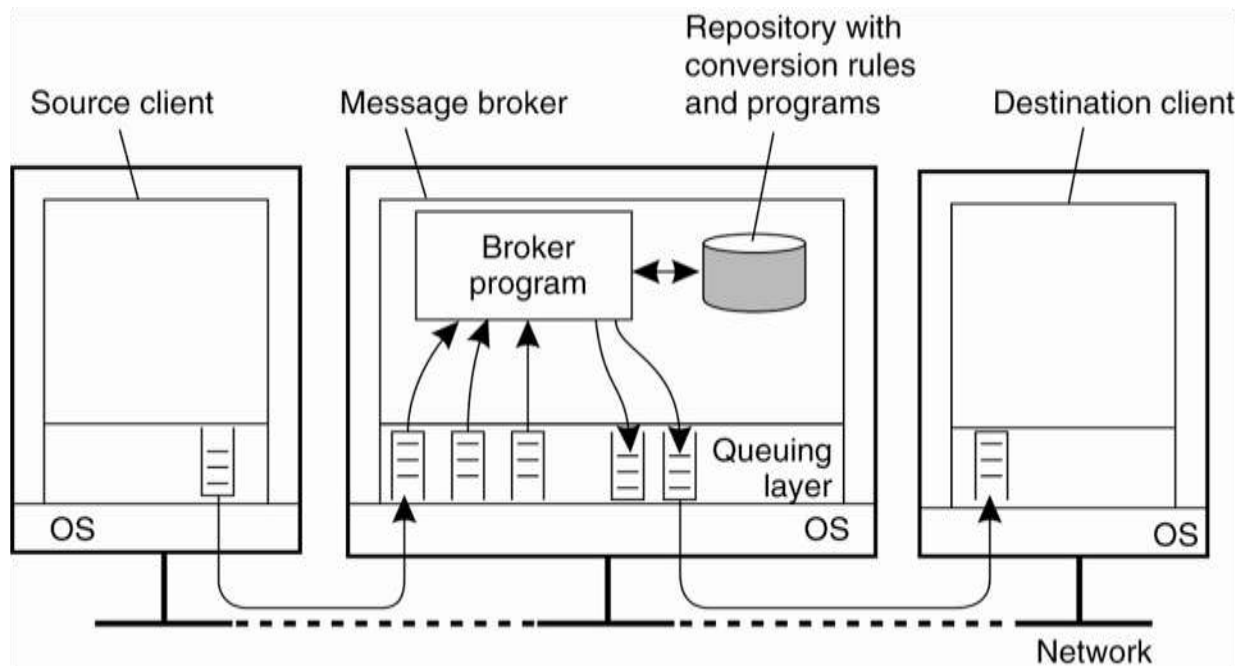
Message queuing: Architectural Issues

- Queues are manipulated by queue managers
 - Local and/or remote, acting as relays (a.k.a. applicative routers)
- Relays often organized in an overlay network
 - Messages are routed by using application-level criteria, and by relying on a partial knowledge of the network
 - Improves fault tolerance
 - Provides applications with multi-point without IP-level multicast



Message queuing: Architectural Issues

- Message brokers provide application-level gateways supporting message conversion
 - Useful when integrating sub-systems



Publish-subscribe

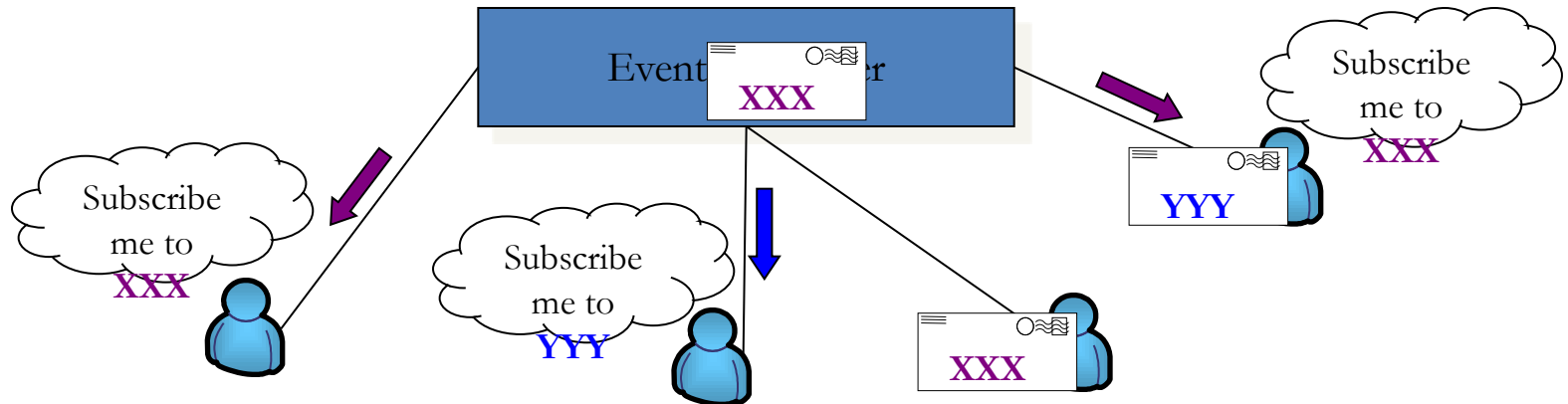
- Application components can *publish* asynchronous *event notifications*, and/or declare their interest in event classes by issuing a *subscription*
 - extremely simple API: only two primitives (*publish*, *subscribe*)
 - event notifications are simply messages
- Subscriptions are collected by an *event dispatcher* component, responsible for routing events to all matching subscribers
 - Can be centralized or distributed
- Communication is:
 - Transiently asynchronous
 - Implicit
 - Multipoint
- High degree of decoupling among components
 - Easy to add and remove components
 - Appropriate for dynamic environments

Subscription Language

- The expressiveness of the subscription language allows one to distinguish between:
 - *Subject-based* (or topic-based)
 - The set of subjects is determined a priori
 - Analogous to multicast
 - e.g., subscribe to all events about “Distributed Systems”
 - *Content-based*
 - Subscriptions contain expressions (event filters) that allow clients to filter events based on their content
 - The set of filters is determined by client subscriptions
 - A single event may match multiple subscriptions
 - e.g., subscribe to all events about a “Distributed System” class with date greater than 16.11.2004 and held in classroom D04
- The two can be combined
- Tradeoffs:
 - Complexity of the implementation vs. expressiveness
 - However, expressiveness allows additional filtering!

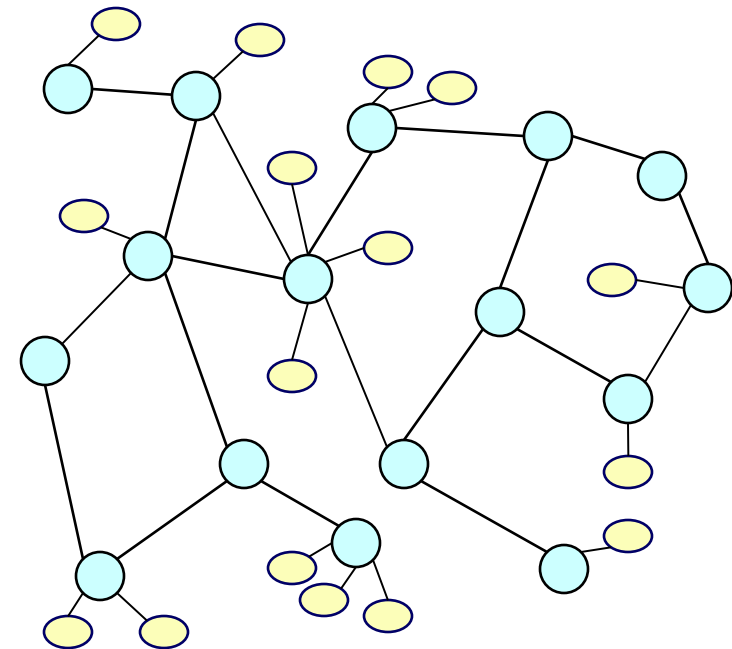
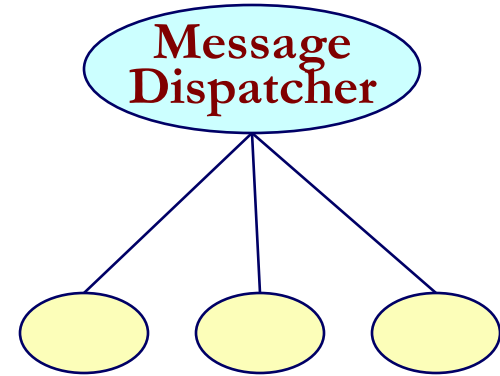
The event dispatcher

- In event-based systems a special component of the architecture, the *event dispatcher*, is in charge of collecting subscriptions and routing event notifications based on such subscriptions
 - For scalability reasons, its implementation can be distributed



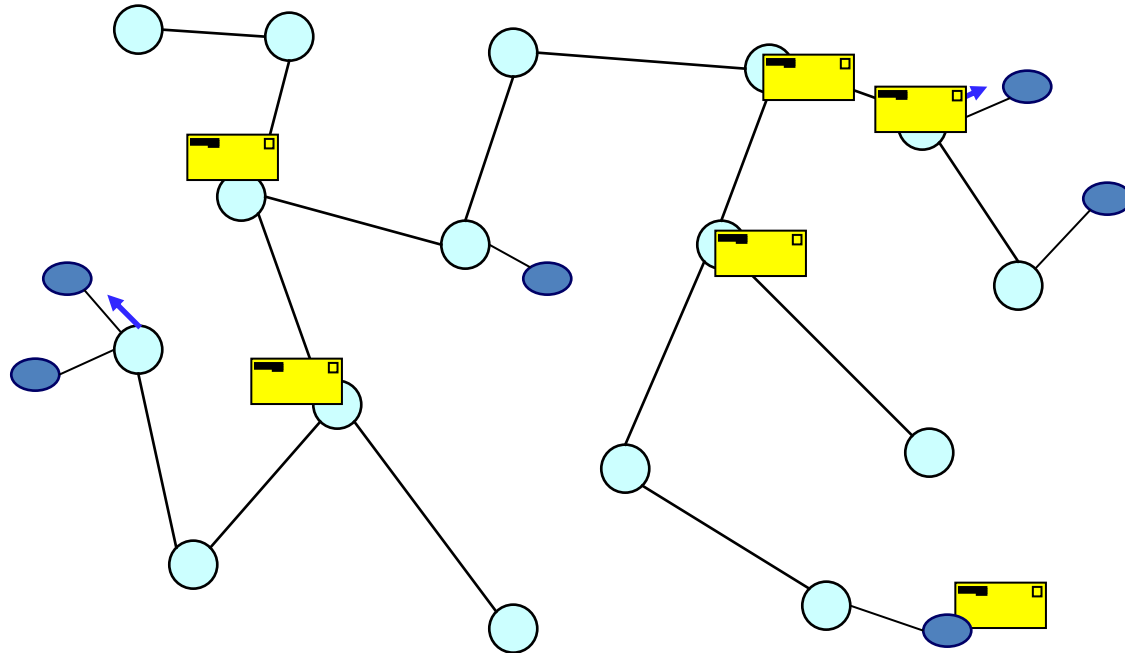
Architecture of the Dispatcher

- Centralized
 - A single component is in charge of collecting subscriptions and forward messages to subscribers
- Distributed
 - A set of *message brokers* organized in an *overlay network* cooperate to collect subscriptions and route messages
 - The topology of the overlay network and the routing strategy adopted may vary
 - Acyclic vs. cyclic overlay



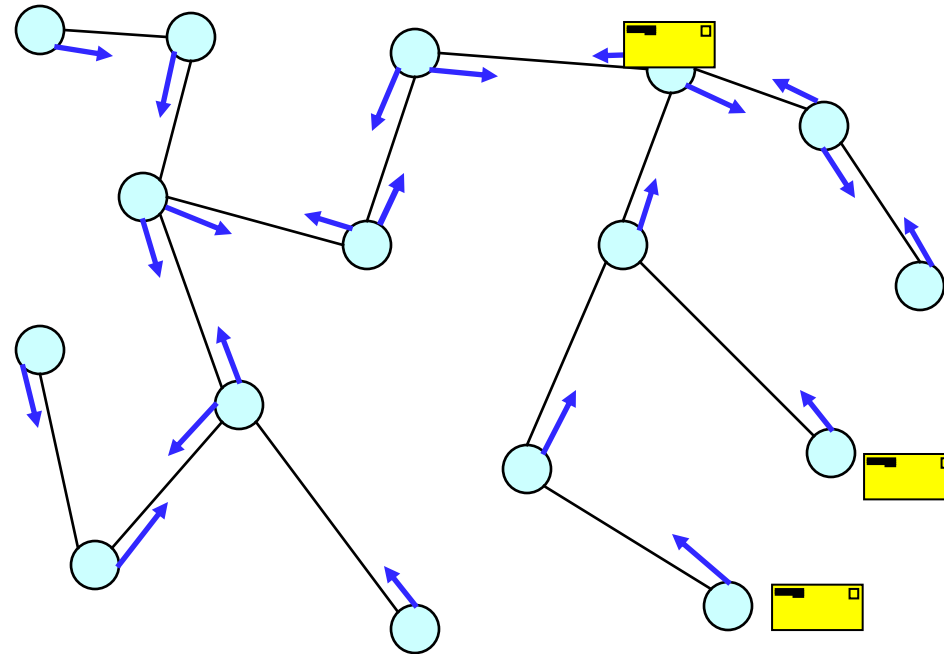
Message forwarding on an acyclic graph

- Every broker stores only subscriptions coming from directly connected clients
- Messages are forwarded from broker to broker...
- ...and delivered to clients only if they are subscribed



Subscription forwarding on an acyclic graph

- Every broker forwards subscriptions to the others
- Subscriptions are never sent twice over the same link
- Messages follow the routes laid by subscriptions
- Optimizations may exploit coverage relationships
 - E.g., "**Distributed ***" > "**Distributed systems**"
 - Fusion, subsumption, summarization



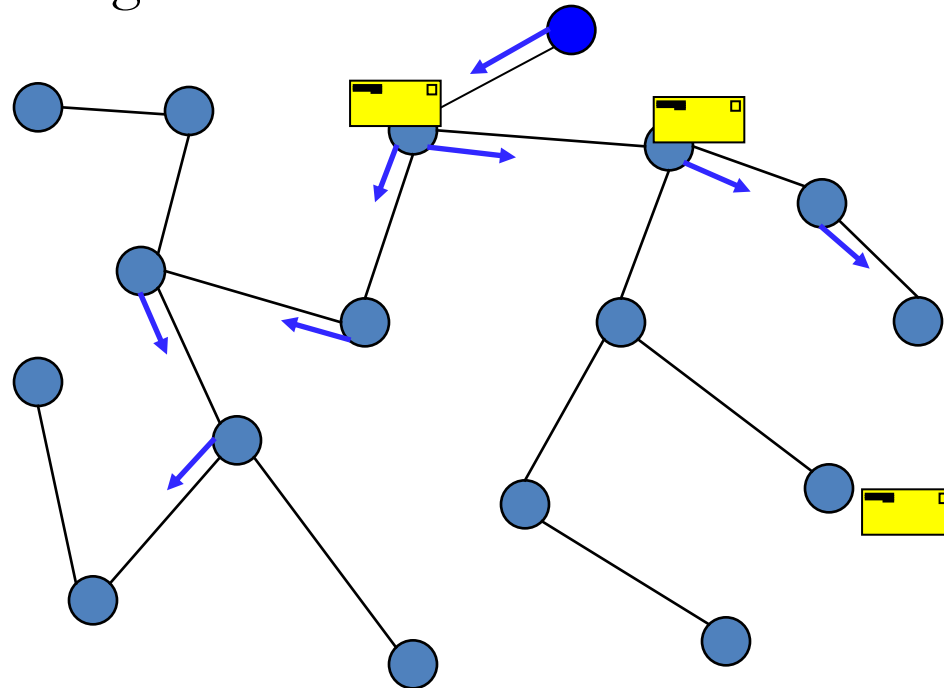
Subscription forwarding on an acyclic graph:

Details

- Each time a broker receives a message it must match it against the list of received filters to determine the list of recipients
- The efficiency of this process may vary, depending on the complexity of the subscription language, but also on the forwarding algorithm chosen
 - It greatly influences the overall performance of the system

Hierarchical forwarding

- Assumes a rooted overlay tree
- Both messages and subscriptions are forwarded by brokers towards the root of the tree
- Messages flow “downwards” only if a matching subscription had been received along that route



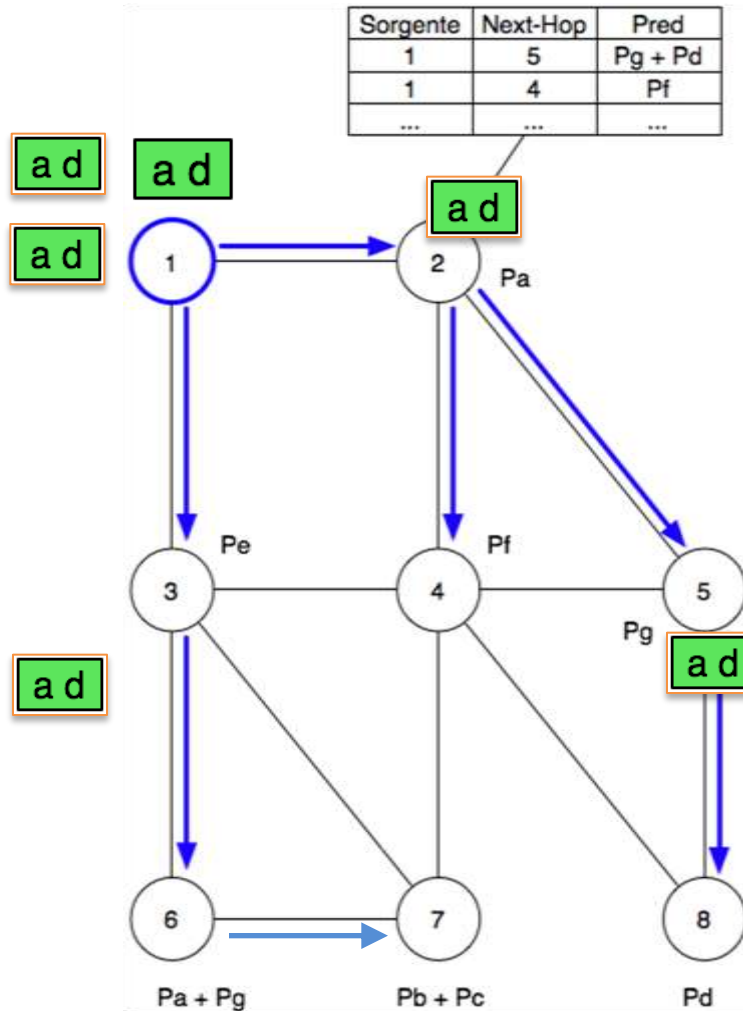
Cyclic topologies: a DHT based approach

- A DHT organizes nodes in a structured overlay allowing efficient routing toward the node having the smaller ID greater or equal than any given ID (the successor)
- To subscribe for messages having a given subject S
 - Calculate a hash of the subject Hs
 - Use the DHT to route toward the node $\text{succ}(Hs)$
 - While flowing toward $\text{succ}(Hs)$ leave routing information to return messages back
- To publish messages having a given subject S
 - Calculate a hash of the subject Hs
 - Use the DHT to route toward the node $\text{succ}(Hs)$
 - While flowing toward $\text{succ}(Hs)$ follow back routes toward subscribers

Cyclic topologies: Content-based routing

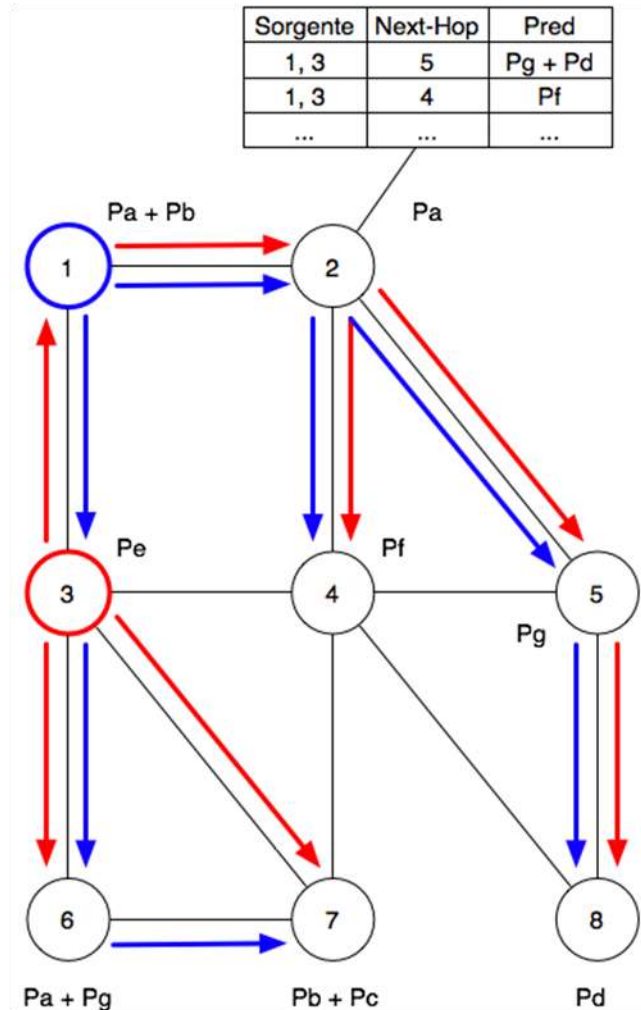
- Useful to differentiate between *forwarding* and *routing*
- Different forwarding strategies:
 - Per source forwarding (PSF)
 - Improved per source forwarding (iPSF)
 - Per receiver forwarding (PRF)
- Different strategies to build paths...
 - Distance Vector (DV)
 - Link-State (LS)
- ... and populate forwarding tables
 - We will skip this!

PSF: Per-Source Forwarding



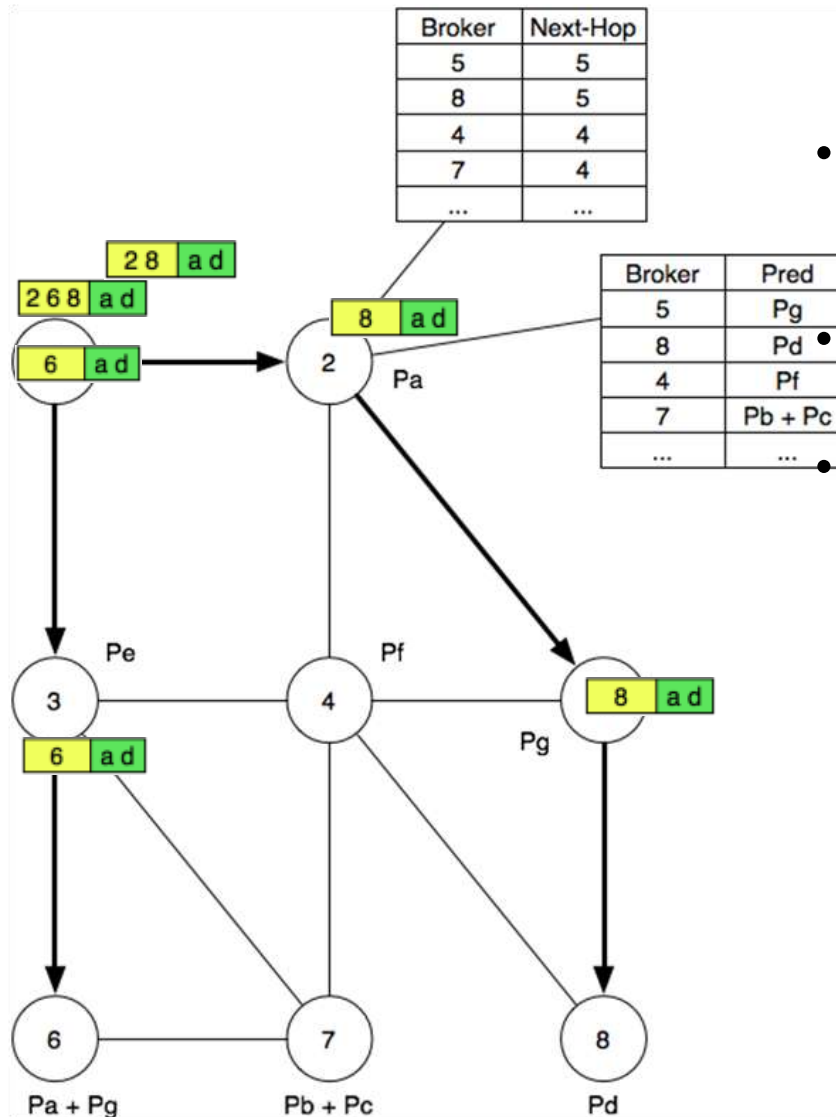
- Every source defines a shortest path tree (SPT)
- Forwarding table keeps information organized per source
 - For each source v the children in the SPT associated with v
 - For each children u a predicate which joins the predicates of all the nodes reachable from u along the SPT

iPSF: improved PSF



- Same as PSF but leveraging the concept of *indistinguishable sources*
- Two sources $A \in B$ with SPT $T(A)$ and $T(B)$ are indistinguishable from a node n if n has the same children for $T(A)$ and $T(B)$ and reaches the same nodes along those children
- This concept brings several advantages
 - Smaller forwarding tables
 - Easier to build them

PRF: Per-Receiver Forwarding



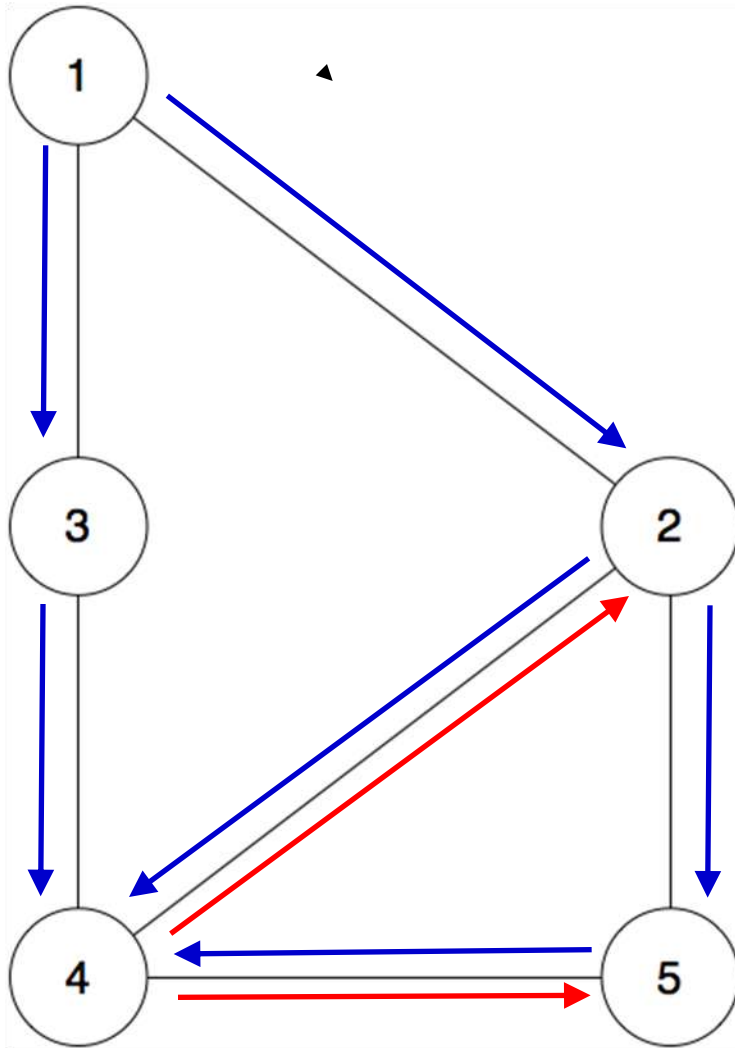
- The source of a message calculates the set of receivers and add them to the header of the message

At each hop the set of recipients is partitioned

Two different tables:

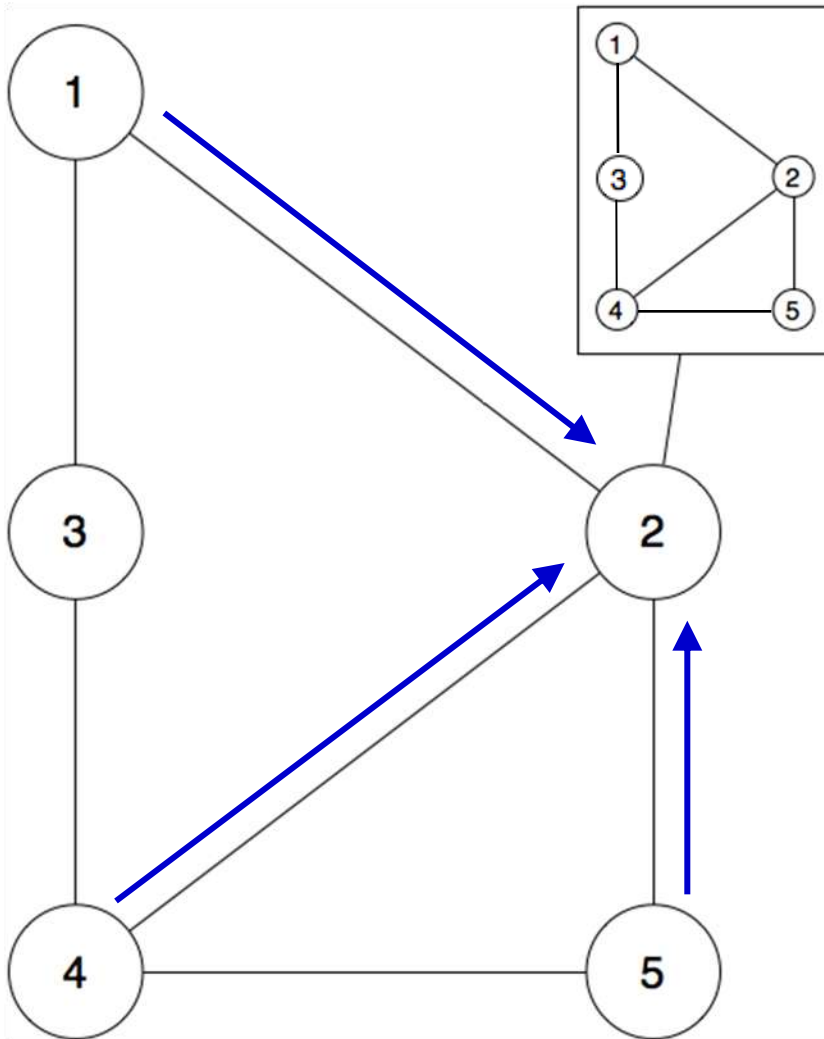
- The unicast routing table
- A forwarding table with the predicate for each node in the network

Distance Vector (DV)



- Builds minimum latency SPTs
- Use request packets (**config**) and reply ones (**config response**).
- Every node acquire a local view of the network (its neighbors in the SPT)

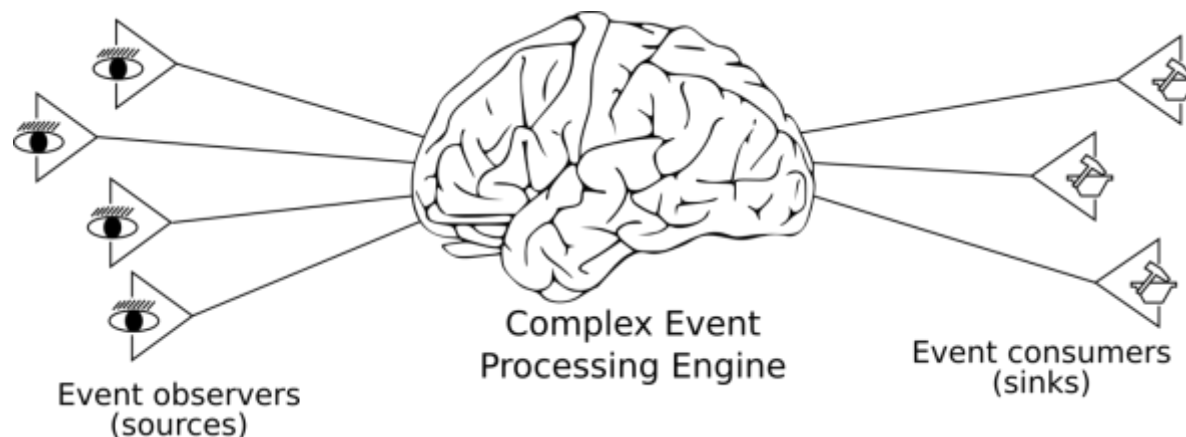
Link-State (LS)



- Allows to build SPTs based on different metrics
- Use packets carrying information about the known state of the network ([Link-State Packet](#) - LSP) forwarded when a node acquires new information
- Every node discovers the topology of the whole network
- SPTs are calculated locally and independently by every node

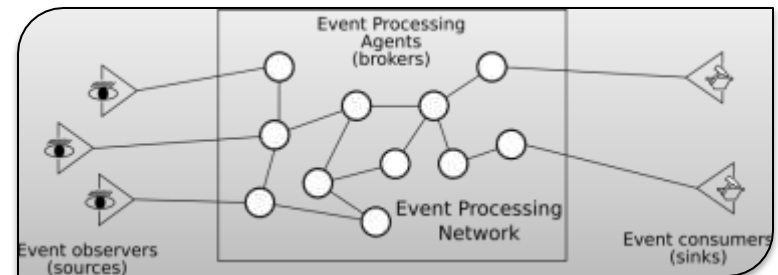
Complex event processing

- CEP systems adds the ability to deploy rules that describe how composite events can be generated from primitive (or composite) ones
 - Recently a number of languages and systems have been proposed to support such architecture (both under the CEP and DSMS labels)



CEP: Open issues

- The rule language
 - Find a balance between expressiveness and processing complexity
- The processing engine
 - How to efficiently match incoming (primitive) events to build complex ones
- Distribution
 - How to distribute processing
 - Clustered vs. networked solutions



Contents

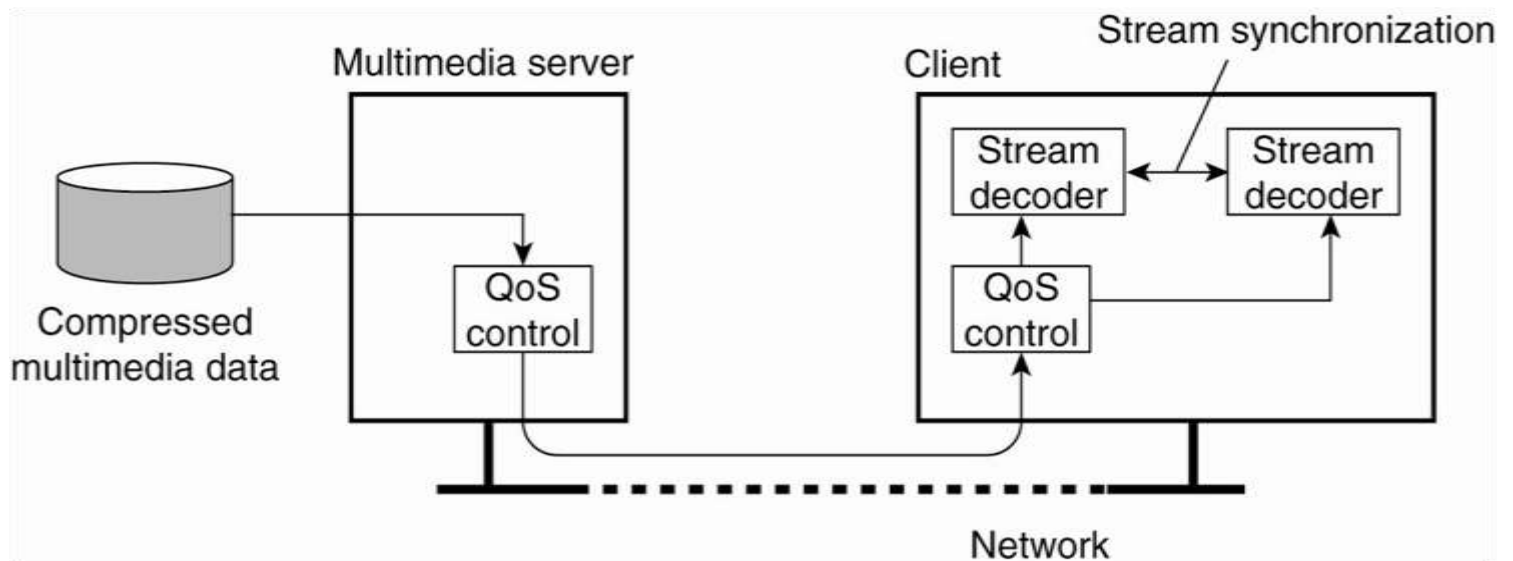
- Fundamentals
 - Protocols and protocol stacks
 - Middleware
- Remote procedure call
 - Fundamentals
 - Discovering and binding
 - Sun and DCE implementations
- Remote method invocation
 - Fundamentals
- Message oriented communication
 - Fundamentals
 - Message passing (sockets and MPI)
 - Message queuing
 - Publish/subscribe
- **Stream-oriented communication**
 - **Fundamentals**
 - **Guaranteeing QOS**

Stream-oriented communication

- Data stream: A sequence of data units
 - Information is often organized as a sequence of data units. E.g., text, audio,...
- Time usually does not impact the *correctness* of the communication
 - Just its performance
- In some cases this is not the case
 - E.g., when sending a video “in streaming”, i.e., to be played “on-line”
- Transmission modes
 - *Asynchronous*: The data items in a stream are transmitted one after the other without any further timing constraints (apart ordering)
 - *Synchronous*: There is a max end-to-end delay for each unit in the data stream
 - *Isochronous*: There is max and a min end-to-end delay (bounded jitter)
- Stream types: Simple vs. complex streams (composed of substreams)

Streaming stored data

- Non-functional requirements are often expressed as *Quality of Service* (QoS) requirements
 - Required bit rate
 - Maximum delay to setup the session
 - Maximum end-to-end delay
 - Maximum variance in delay (jitter)



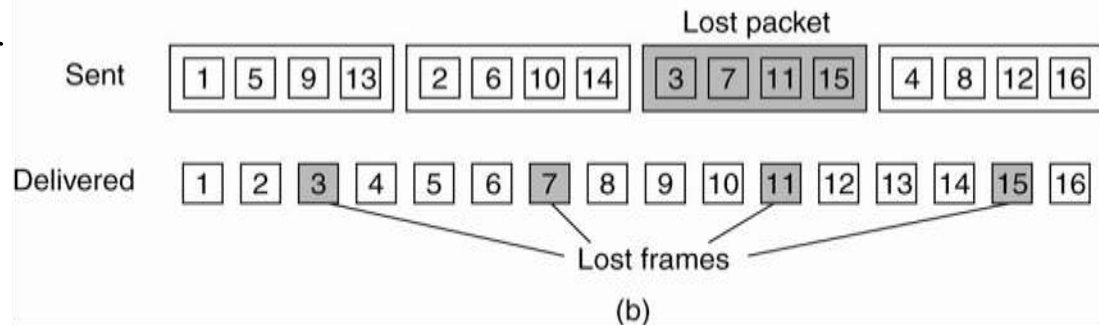
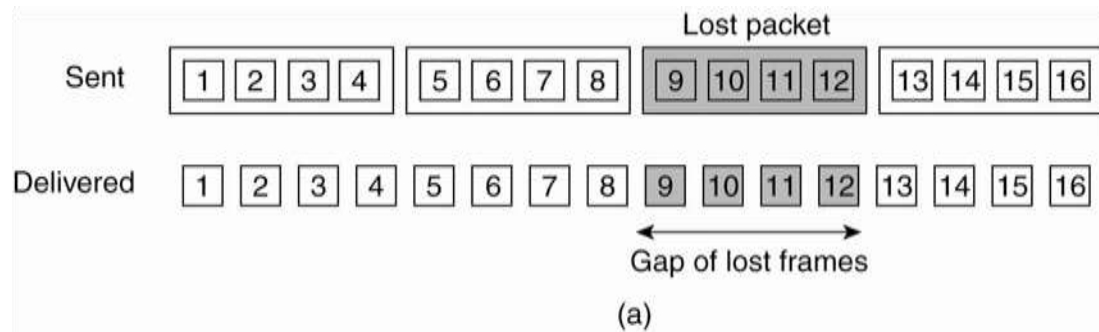
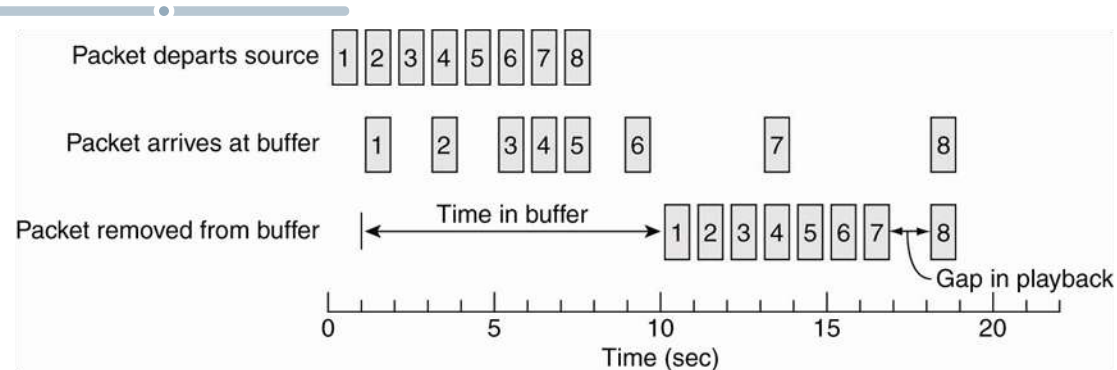
QoS and the Internet:

The DiffServ architecture

- IP is a best effort protocol!
 - So much for QoS :-)
- Actually it offers a Differentiated Services field (aka Type Of Service - TOS) into its header
 - 6 bits for the Differentiated Services Code Point (DSCP) field
 - 2 bits for the Explicit Congestion Notification (ECN) field
 - The former encodes the Per Hop Behaviour (PHB)
 - Default
 - Expedited forwarding. Usually less than 30% of traffic and often much less
 - Assured forwarding (further divided into 4 classes)
- Not necessarily supported by Internet routers

Enforcing QoS at the application layer

- Buffering
 - Control max jitter by sacrificing session setup time
- Forward error correction
- Interleaving data
 - To mitigate the impact of lost packets



Stream synchronization

- Synchronizing two or more streams is not easy
- It could mean different things
 - Left and right channels at CD quality → each sample must be synchronized → 23 μ sec of max jitter
 - Video and audio for lip synch → each audion interval must be in synch with its frame → at 30 fps 33 msec of max jitter
- Synchronization may take place at the sender or the receiver side
 - In the former case the different streams can be merged together
- It may happen at different layers (application vs. middleware)

