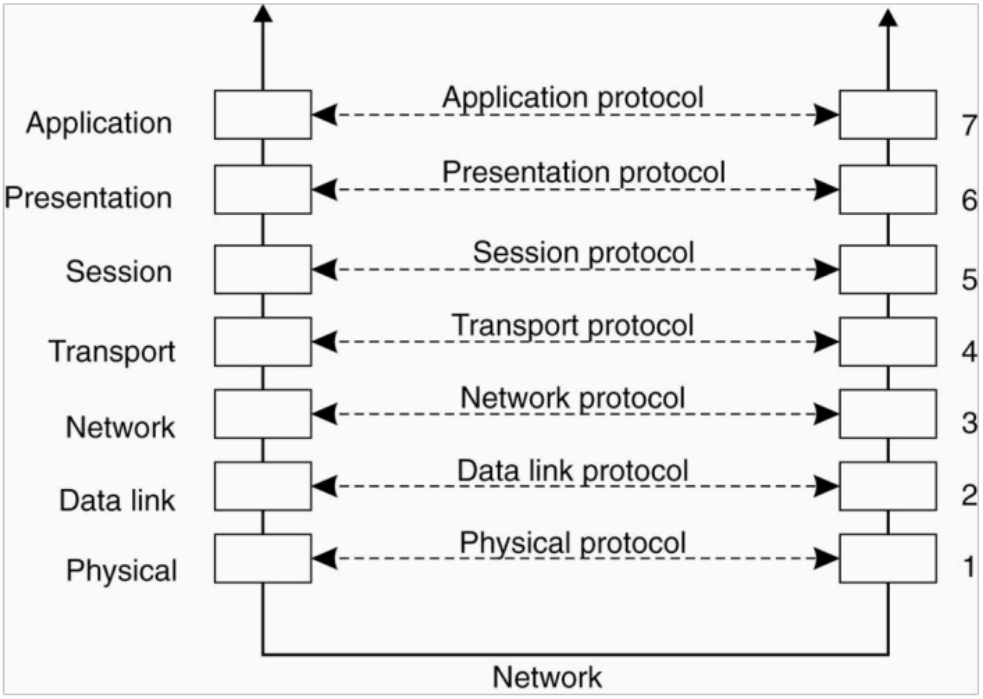
03.Communication

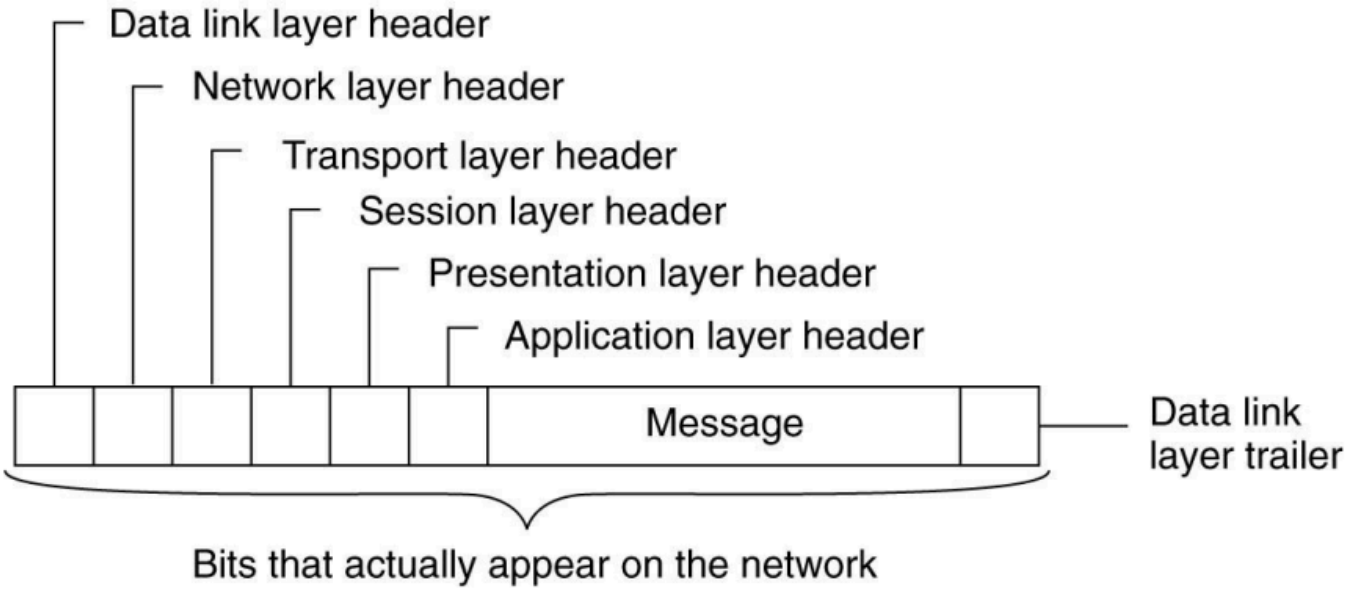
Describing the general way to distribute messages in the network, we are not going to describe technologies but the general models.

Fundamentals-Protocols and Protocol stacks, Middleware

Layered protocols: the OSI model



Each protocols take advantages of protocols that are lower of it in its stack. In order to classify the protocol we use the service they offer at each layer.
Physical protocol design how the connections between nodes is actuated.
Data Link protocols govern the rules that permit to exchange bits or bytes between two nodes.
The network protocol is a key protocol that allows real distributed architecture to work as it introduces the rules that permit the communication.
Transport protocol permit a powerful communication.
We don't distinguish between Session, Presentation and Application layer in Internet and we refer to them as Application protocols.
The internet is based on Network and Transport protocols. We can place the middleware on top of the Transport layer and in it we can put the applications.
The practical way protocols work is by using a method called encapsulation

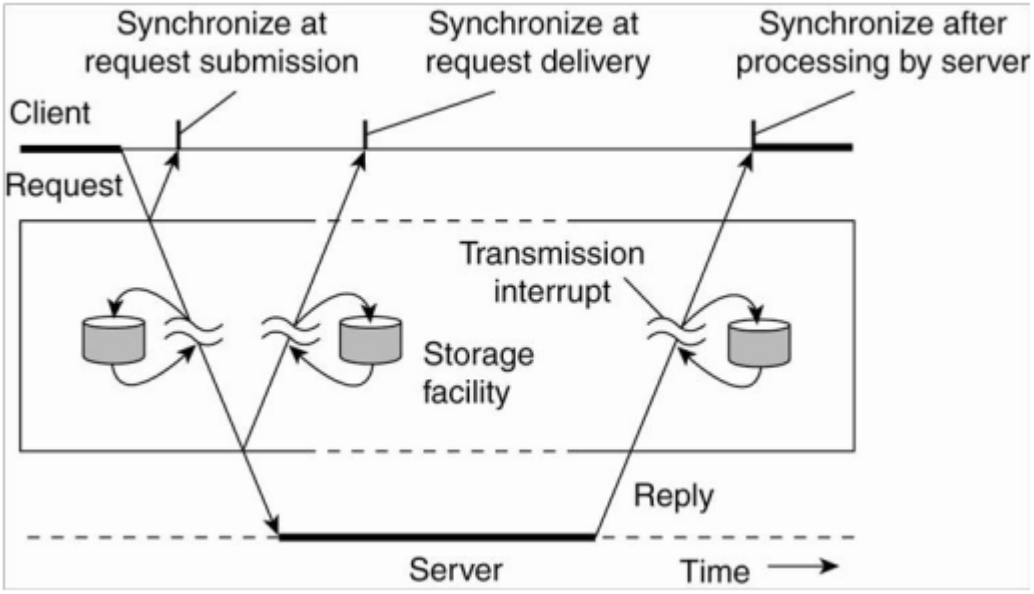


The protocol put an header to the message and send it to the protocol that is lower than it. At each layer you take the message on the layer on top of you, it becomes the body of your message and put an header on it. Middleware does a similar thing putting functionalities on top of the Application layer.

Types of Communication

Middleware may offer different form of communication:

- Transient(receiver and sender needs to be active at the time of the communication) vs. persistent(receiver and sender are decoupled in time, the message is sent and read at different time), both are decoupled in space
- Synchronous(strong synchronisation between sender and receiver, usually transient) vs. asynchronous (various forms, not connected at the same time)

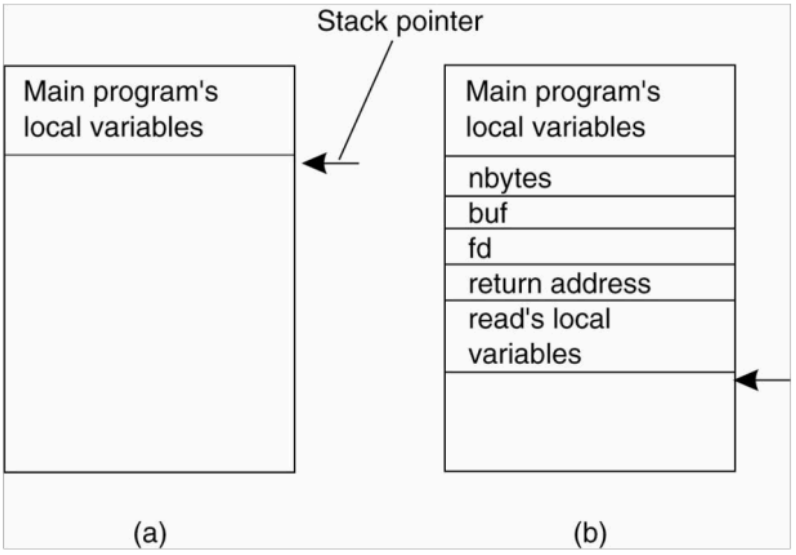


Remote procedure call-Fundamentals, Discovering and Binding, Sun and DCE Implementations

It is a form of communication that tries to mimic standard procedure code in a network.

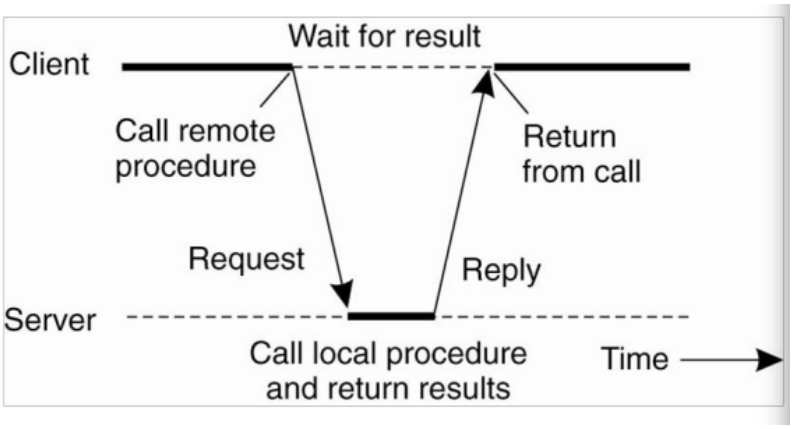
Local procedure call

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

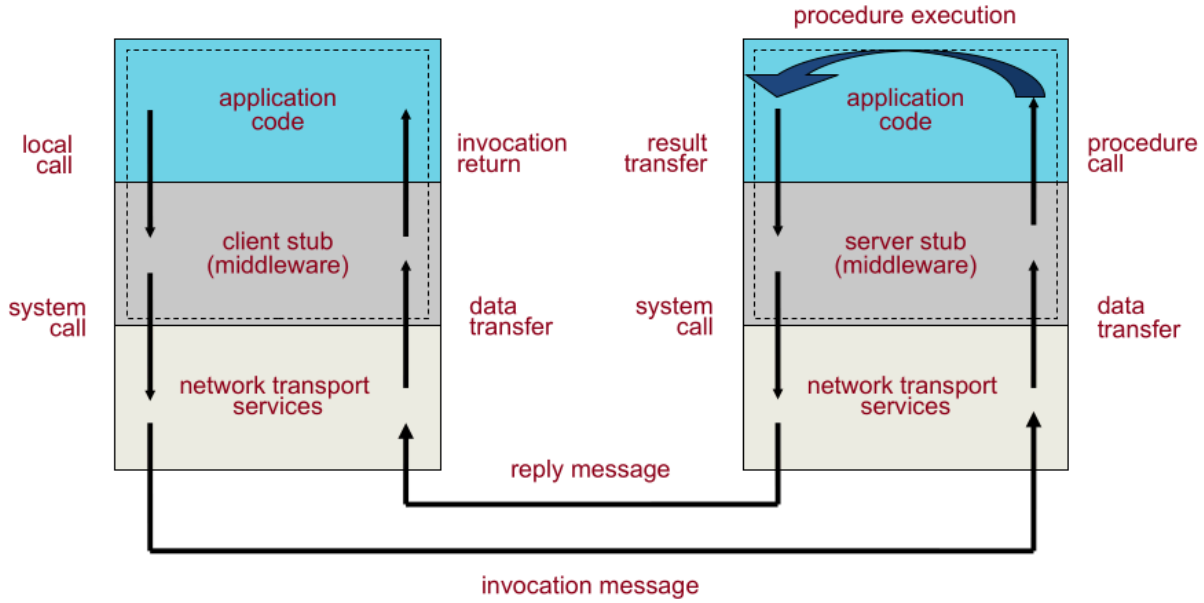


Only at the end of this process the code can start its execution.
We can pass parameters to a procedure with different mechanisms:

- By value: like C code, the content of the local variable is copied in the stack. Used by 99.9% of procedural languages
- By reference: like C code, the invoker and the inokee are sharing the space where the data reside as they are using a pointer to it
- By copy/restore: at the start the actual parameter are copied in the formal parameter, the restore act at the end of the execution as the value that the formal parameter assume are copied in the actual parameter. It breaks aliasing, every code impacted by it works differently.
  In general we want something that works like it is working locally but is remote: the client start a procedure, send all the data to the server and wait that the server end the execution before continuing its execution.

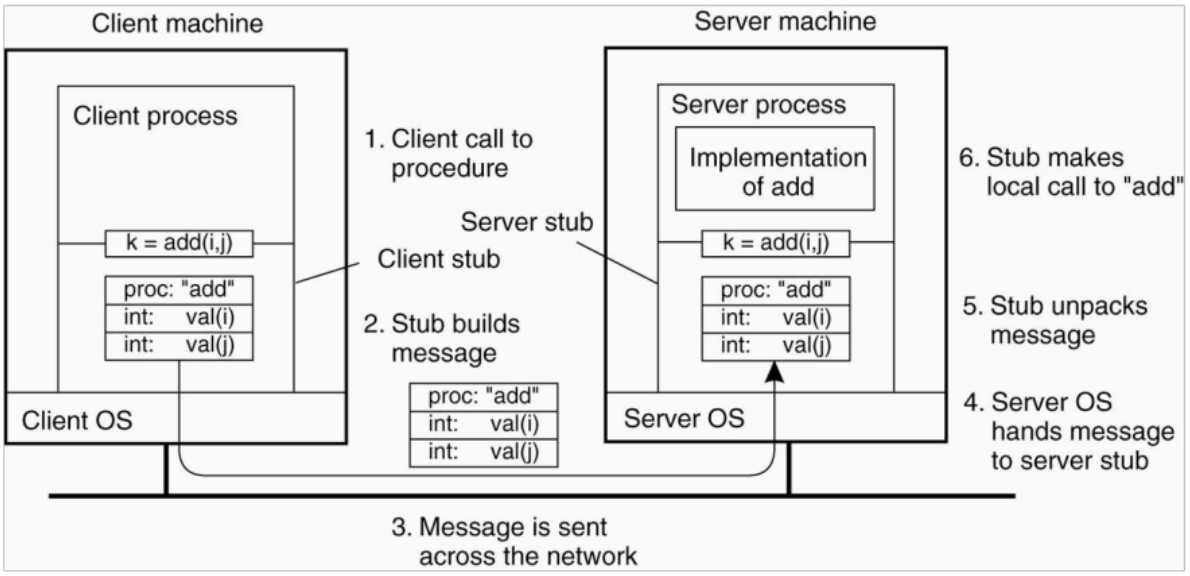


RCP: How it works



You invoke a local procedure with the same parameters needed for the procedure but instead of executing it transform the data in a message that is sent to the middleware to be executed and then the result is sent to the client. The client middleware flatten the message in a stream of bytes, also flatten the complex data structures. The first flatting is called serialization and marshaling is the opposite procedure done by the server middleware as it transform the stream of bytes in data that can be read by the server procedures.
If there are errors you get Exceptions(usually in local procedure you get crashes) so you need to cope with exceptions

PARAMETER PASSING BY VALUES



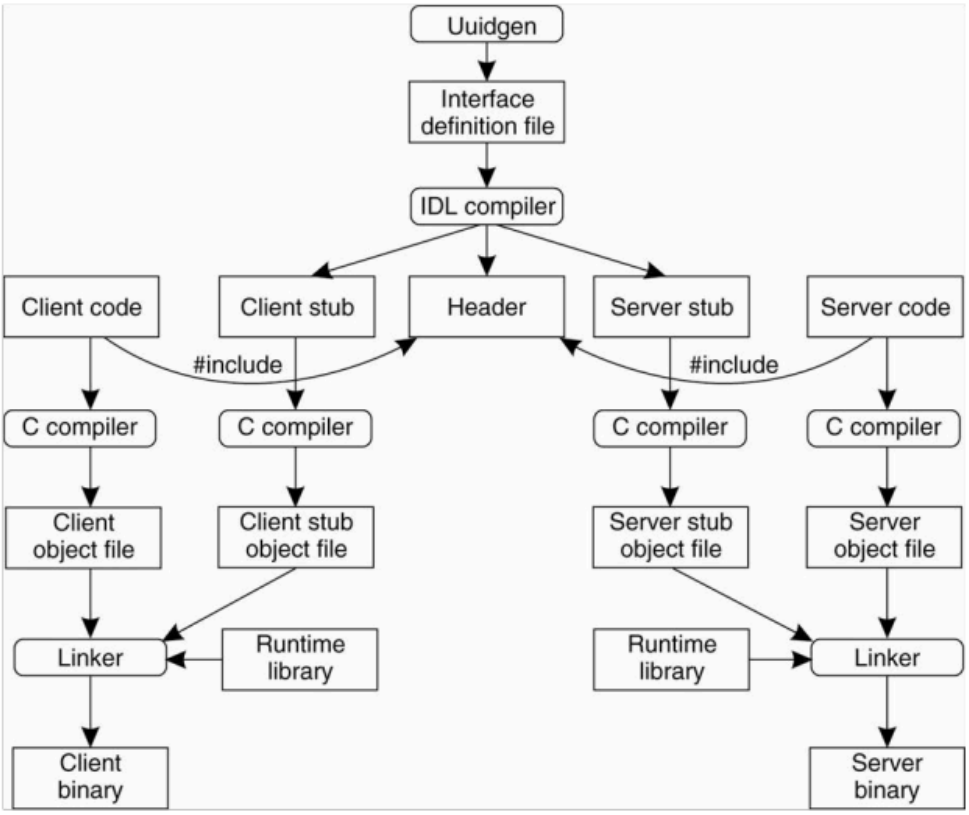
Passing by reference in RPC is pratically impossible as you should recreate the memory on the server and doing so you are basically doing passing by copy/restore. The middleware can recreate the code only if it know the signature of the procedure.

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
  Must be introduce to define the signature of the procedure. It is usually a different language from the one of the procedure.

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

Binding Client to the Server

The client middleware need to know how to link the local procedure to the remote procedure. We need to define the binding by the client and the actual serve. The first problem is to find where the software in the server is running and bind it to the server. A solution is to write the application as it is centralised but you run it distributed(usually not true). You need to decide which machine is running the software and then you can run it infinitely. Once you found the machine you have to identify between several process which one is the one you need.

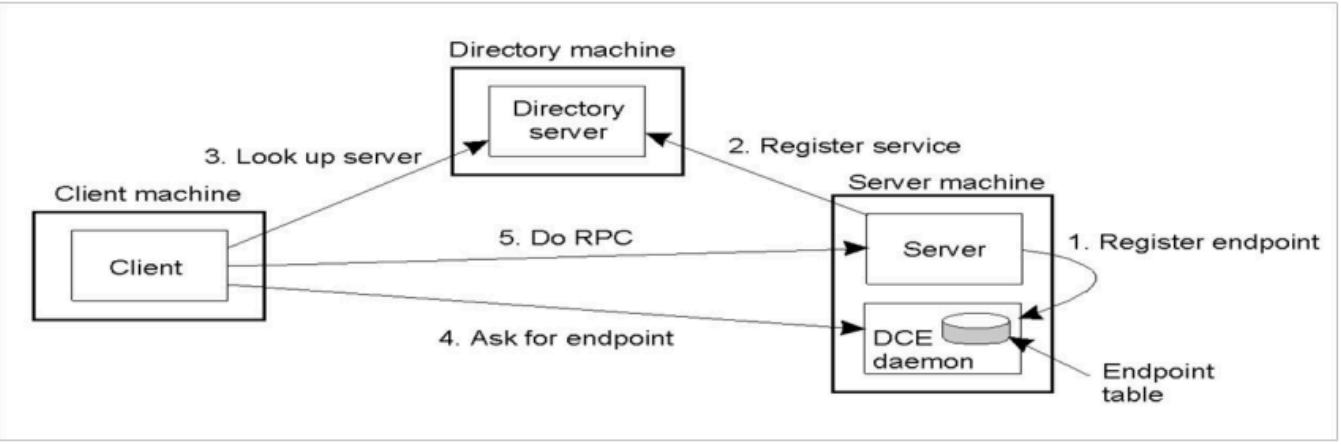
Sun's solution

Resolve the first problem with portmap: the server code permit to inform the middleware that the procedure is ready to run. The choosing of the machine is resolved providing the IP of the machine.

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

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
  It invoke the OS, tell the byte stream and then put it in a shared memory and then the other procedure is invoked.
  It is at the basis of DCOM and .NET, at the base of cut/copy procedure.

Asynchronous RPC

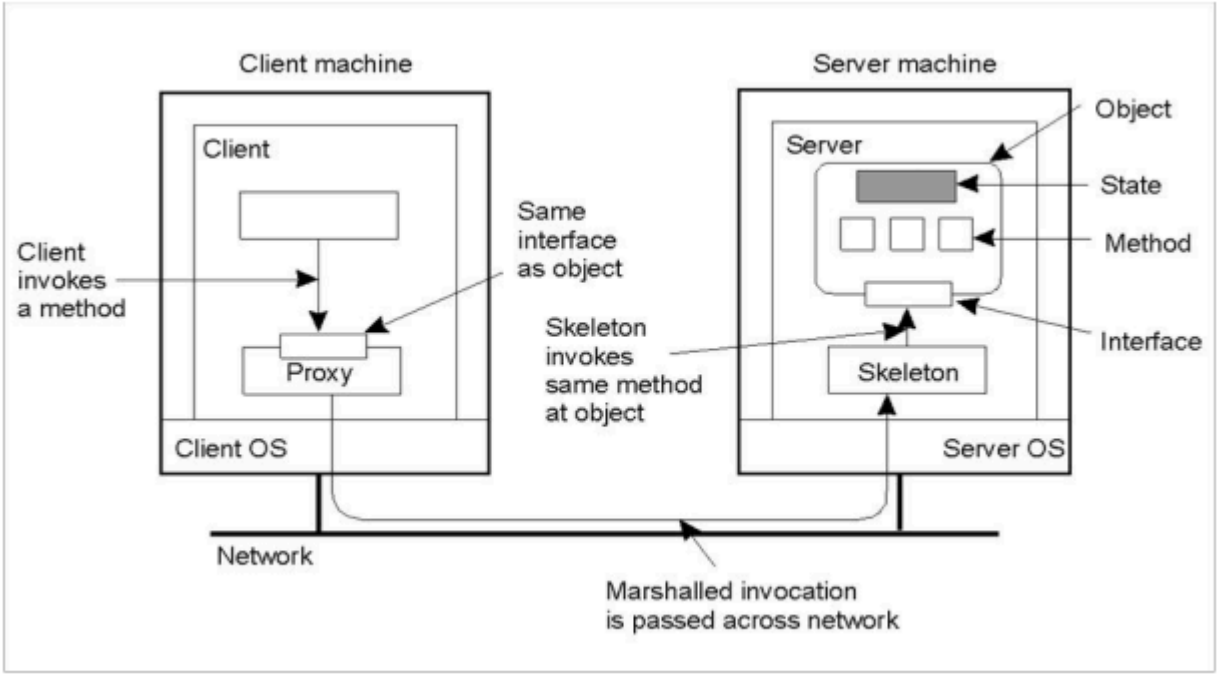
Can only be applied to void procedures. It is nice because can improve throughput.
RPC preserves the usual call behavior

- 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

Remote Method Invocation

Ability to invoke methods of objects through the network. We want to simulate the invocation of methods as in a centralized system. We want simulate in a distributed system a centralized application made of objects, the objects can be passed around by copy.



Communication

We want to connect the client and server objects in a way where the distribution system is as transparent as possible, we want to simulate the fact that there are only local objects.
The middleware makes acknowledgement with a proxy that has the same prototype of the method of the remote objects but its implementation only move the parameters on the other side of the network. On the other side the skeleton receive the parameters and invoke the local methods to execute the required services. The interface of the object is the most important part to automatically generate the proxy.
When we pass objects we need to have a copy of the attributes and the methods.
Problems arise when the server and the client use a different programming

language(you cannot pass around the code and so you cannot pass by copy). We can simulate the passing by reference we don't serialize the code but we transport a proxy to the object needed(Same interface of the house but made to call the methods at the original object). The problem is that this method easily loose control of the situation as you will have multiple recall to methods through the network. You also can have only objects with private attributes as that ensure that the access to memory is done through methods permitting the rewriting of such methods to access memory on different machines.
How it is done 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

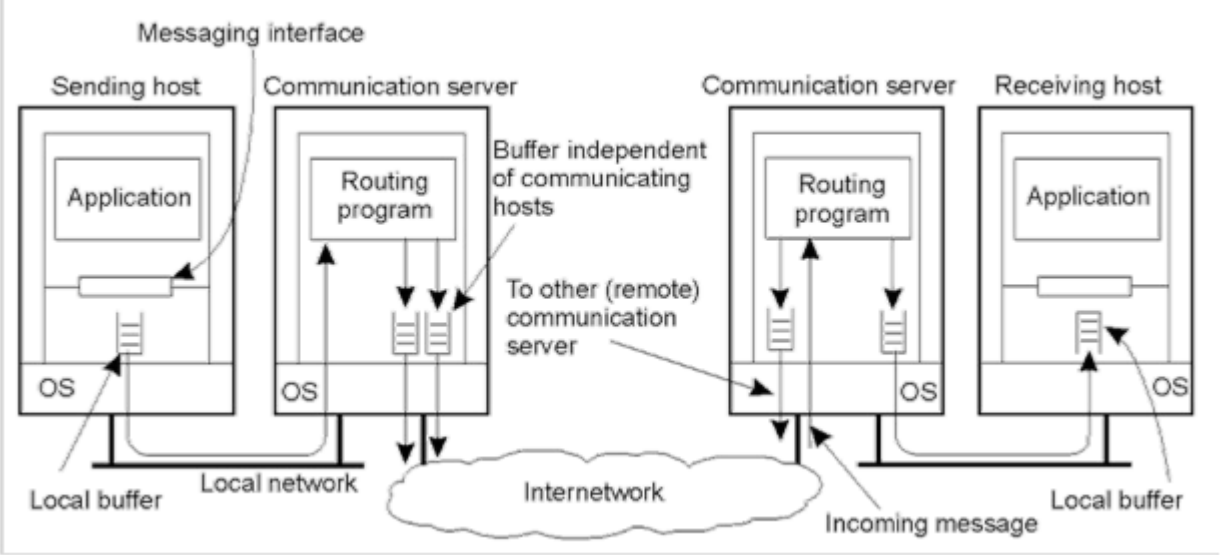
Message Oriented Communication

It is a lower level abstraction world. It is centred on that notion of one-way message/event. It is usually asynchronous. It is point to point. Intrinsically there is a strong connection between caller and callee. Brings more decoupling among components. Often support persistent communication(middleware ensure that a message is correctly received). Often supporting multi point interaction.

Reference Model

All the reference middleware are implemented this way:

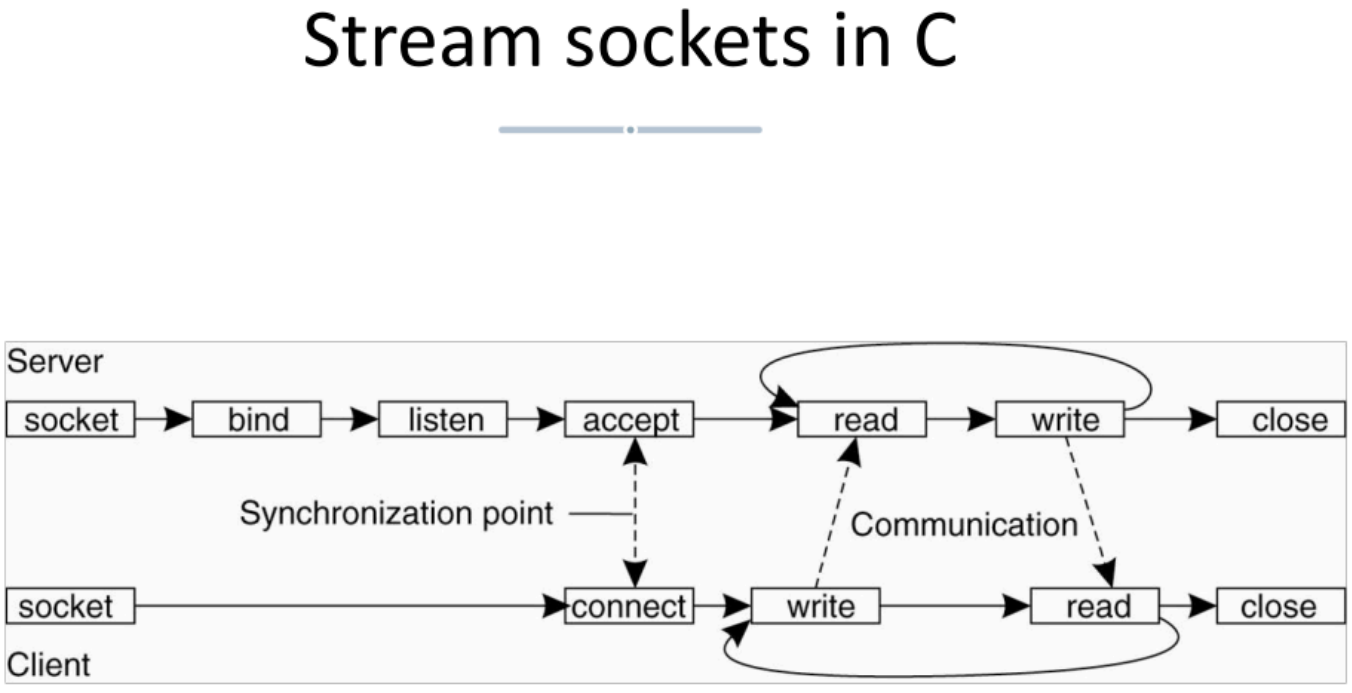
- Sender
- Receiver
- One or more elements that can run on the same machines or numerous machine and reassure that the message will arrive the the receiver



The protocol used is accessed by its API in BSD sockets(generic interface, can open also Unix sockets).

Stream Sockets

Differentiate between server and client. The server sockets accept the connection from the client. From the client you connect 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. A single port is enough as the connection is marked with the combination of server IP and Port and client IP and the port it is using(usually is chosen randomly and it is chosen differently for different connection even if the machine is the same).



If someone close the socket if the other on wants to write on the socket it will receive an exception.

Datagram sockets

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.
It is a connectionless protocol, it not guarantee the delivery of the packet but it tries its best to deliver on time and with the maximum dimension possible.

Multicast Socket

It hides multiple details, have groups and if the multicast is open you can send messages to the group even if you aren't part of the group a closed multicast permit the sending of a message only if you are in the group. Most routers don't route multicast packets out of the LAN.
Socket Pros:

- Found on every OS
  Socket DIsadvantages:
- Low level
- Protocol independent, we need higher level primitives for asynchronous, transient communication.

MPI

Form of message oriented communication at low level but meant for a well connected group of computer that are connected to the same datacenter and want to do high performance communication. Each group has its identifier and each process has its identifier. It permits a lot of control on the communication and the way the protocol work.

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_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

- MPI_bsend wait until the message you send is stored on a buffer and then taken care by someone. When the send return you can reuse the array of the buffer. This not assure that the message is correctly received.
- MPI_ssend: guarantee that the message is received by the other machine and the process has started
- MPI_isend: wait to reuse the buffer until the process is terminated.
- MPI_issend: is like MPI_ssend but is asynchronous
  There is also multicast for entire groups. With reduce you send an array of data, specify the reuse operation, specify that the array must be equally divided between the machines in the group and each machine give its results and the return message contains the sum of each return value from the group.

Message Queuing

Point-to-point persistent asynchronous communication

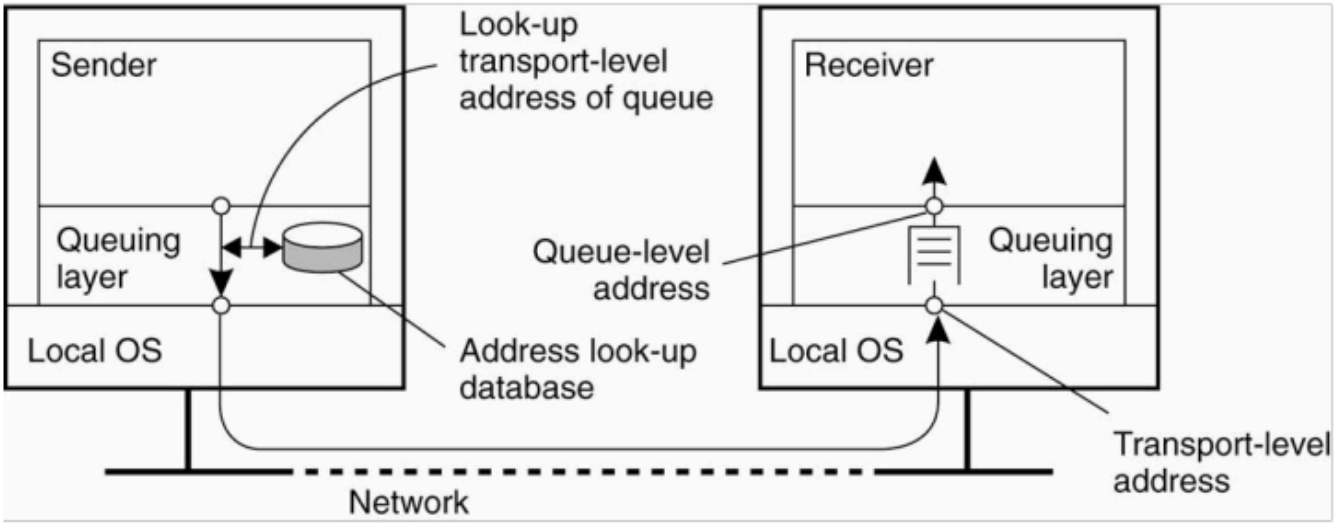
- Typically guarantee only eventual insertion of the message in the recipient queue (no guarantee about the recipient's behaviour)
- Communication is decoupled in time and space
- Can be regarded as a generalization of the e-mail
  Intrinsically peer-to-peer architecture
  Each component holds an input 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

In a queue messaging system you send on the queue. It match the interpretation as datacenter as you put/take from a place(the queue). You can also be notified from the queue.
Queues are persistent and decouples senders and receivers in space and in time(they are persistent). The queue are bad as they are more complicated than RMI or TCP, there is a lot of decoupling. The advantages derive from the fact that they remove the need for the client to remain connected and the queue sharing simplifies load balancing, each request will be processed by only one server.
Queues are identified by symbolic names

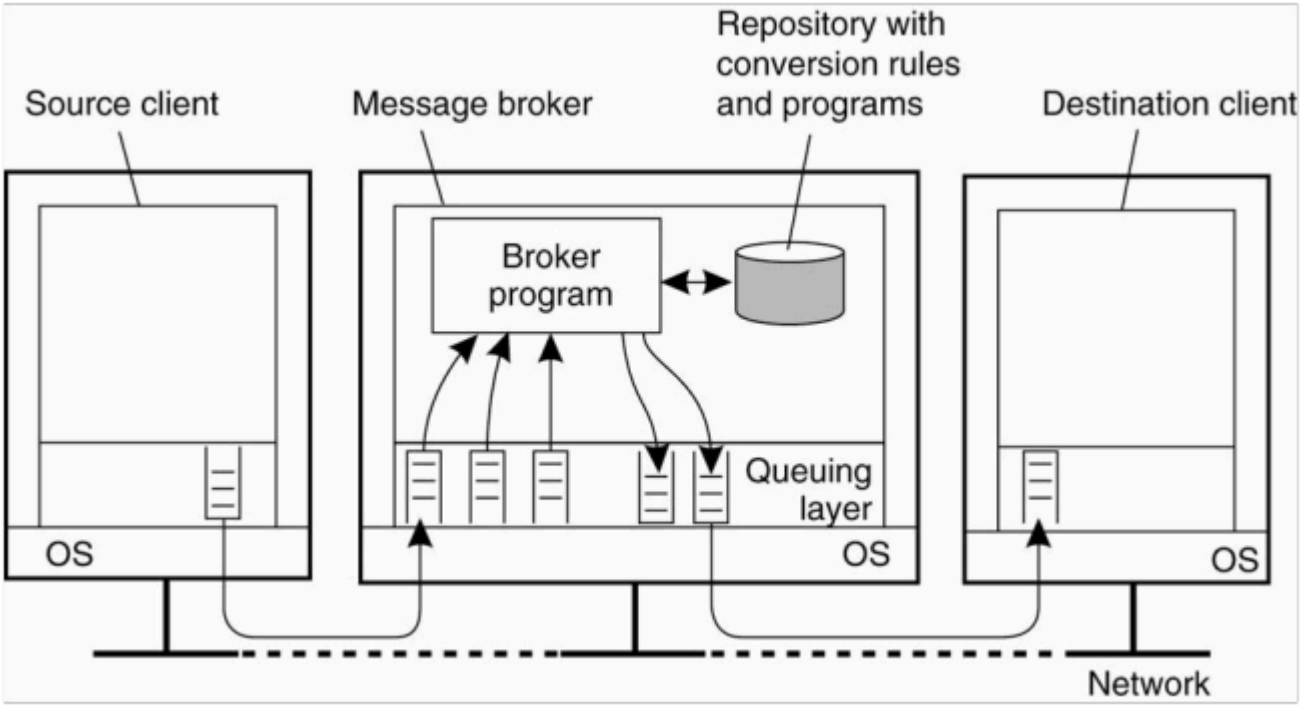
- Need for a lookup service, possibly distributed, to convert queue-level addresses in network addresses
- Often pre-deployed static topology/naming



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



Publish-Subscribe

Typically the event based architecture is implemented by a publish subscribe procedure: procedure that have two actions subscribe and publish

- Subscribe: express interest on something and want to be notified of everything is send on this topic. Based on topic or content. There is a set of topics to choose from. Similar to multicast as the address is the topic and the subscriber are the address of the interested. It is asynchronous and multicast. There is also content based where the content of the message is the thing you are subscribed to. You use feels to match the messages to the subscribers. Subscriptions contains expression(filters) that allow clients to filters events based on their content. This increase the complexity in the middleware
- Publish: address is multicast and depends on the data you are processing
  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

The dispatcher persist the subscription but not the messages-
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.
The message is forwarded to every broker that check its subscription table and decide if it needs to be forwarded.
The cost of subscription is low(low cost in processing on the node) but the message transfer is costly.

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
  On the network you use minimal path, but on the broker you need more power as the subscription tables are going to be big.
  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

You look at the acyclic graph as it is a 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

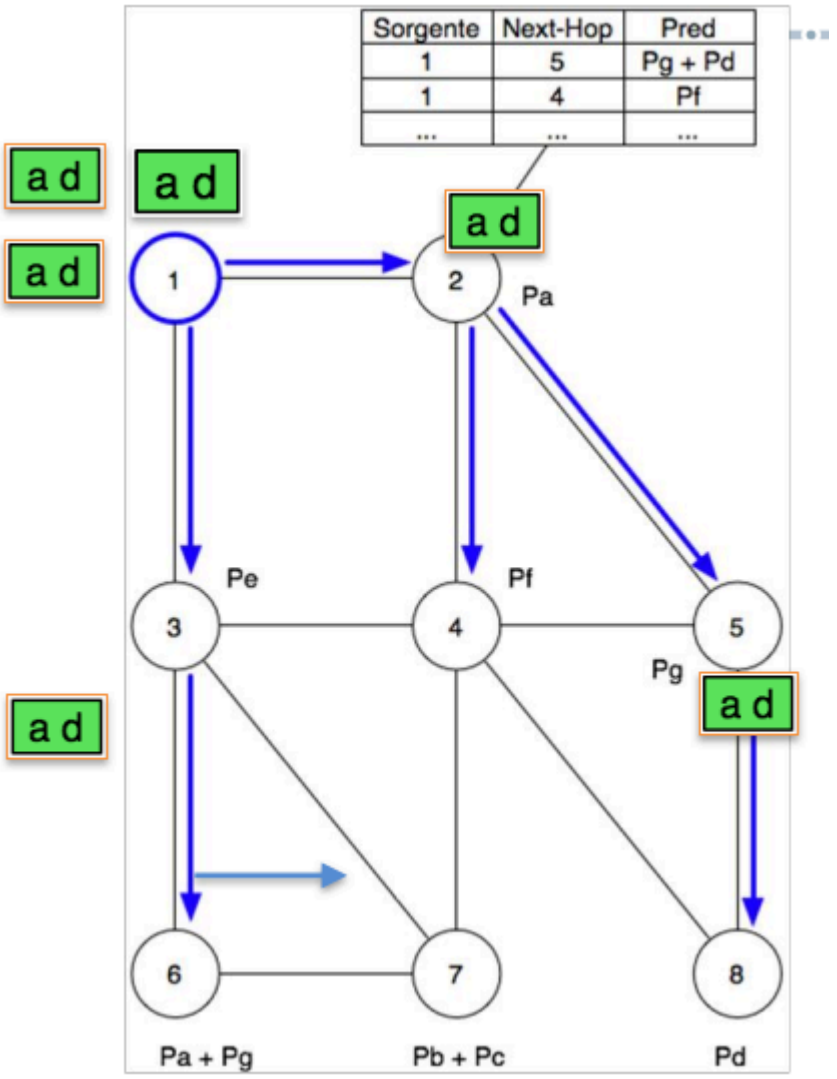
You risk to create a cycle in the subscriptions.
Useful to differentiate between forwarding and routing
Different forwarding strategies:

- 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

PSF: Per-Source Forwarding

Every source defines a shortest path tree (SPT). Forwarding table keeps information organized
per source:

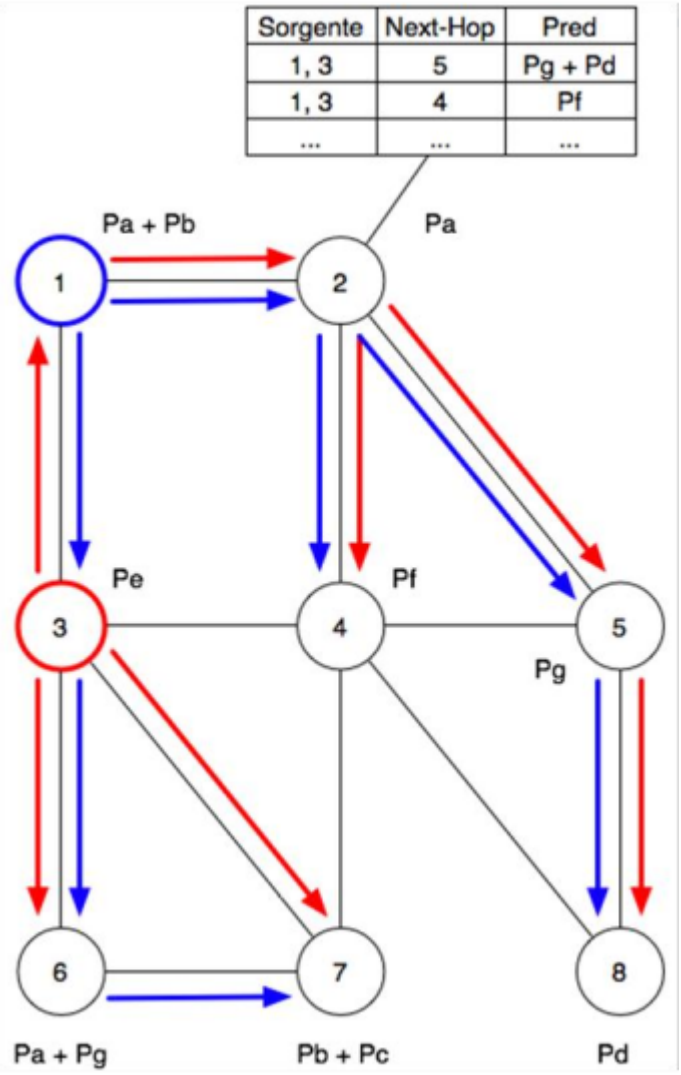
- For each source v the children in the SPT associated with v
- For each children u a predicate which joins the predicates of all the nodes reachable from u along the SPT



iPSF: improved PSF

Same as PSF but leveraging the concept of indistinguishable sources
Two sources A e 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



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 succ(Hs)
- While flowing toward 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 succ(Hs)
- While flowing toward succ(Hs) follow back routes toward subscribers
  They can easily implement public/subscribe mechanism as you link a topic to a key and the subscriber search for this key

Complex event processing

Typically unicast, for each message there is a single node to receive the message.
The idea is to put intelligence into the middleware. You can express rules about the messages, describe how composite events can be generated from primitive (or composite) ones
The rule language can be simple or complex. Subscription can be about raw messages or complex messages generated by as rule when a message entered the broker.

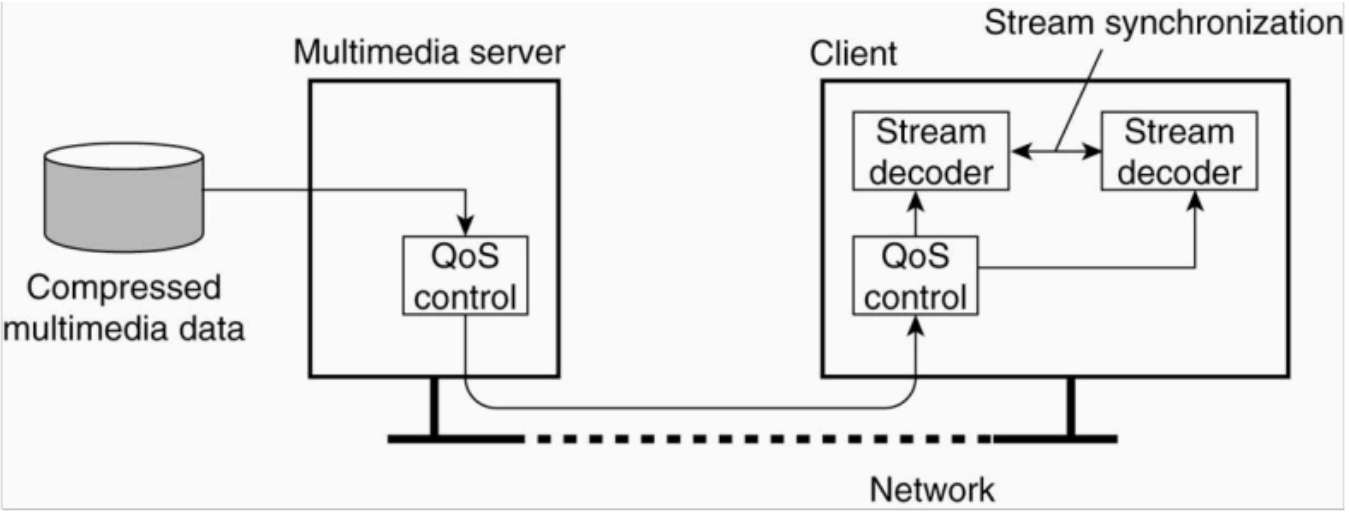
Open issues

- 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)

Stream-oriented communication

The idea is timing of messages are in a sequence/stream of data that compose the implemented service and the time in which you receive the message is critical for the performance and correctness of the service.
Typically the transmission is:

- 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)



Non-functional requirements are often expressed as Quality of Service(QoS) requirements, the client usually decide the QoS but the server can impose some

constraint to maintain a good service as it knows better how much you can stress the network(continuous agreement)

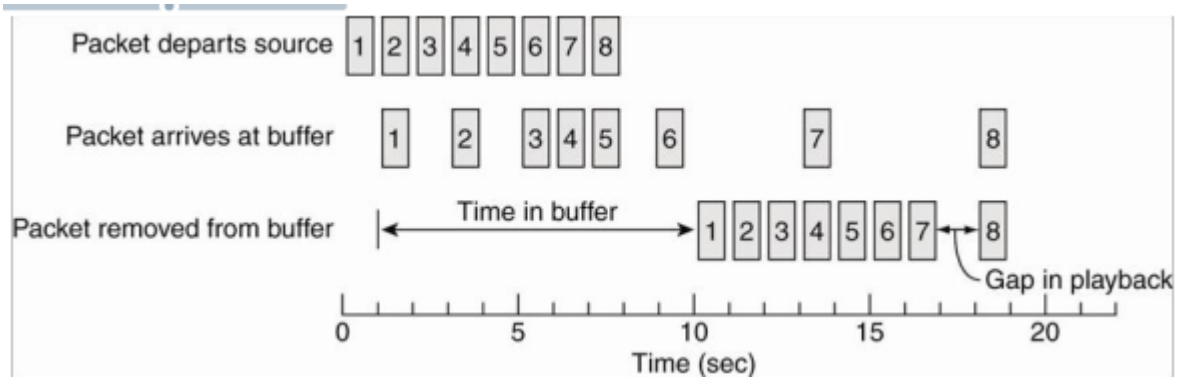
- Required bit rate
- Maximum delay to setup the session
- Maximum end-to-end delay
- Maximum variance in delay (jitter)

QoS and the Internet:The DiffServ architecture

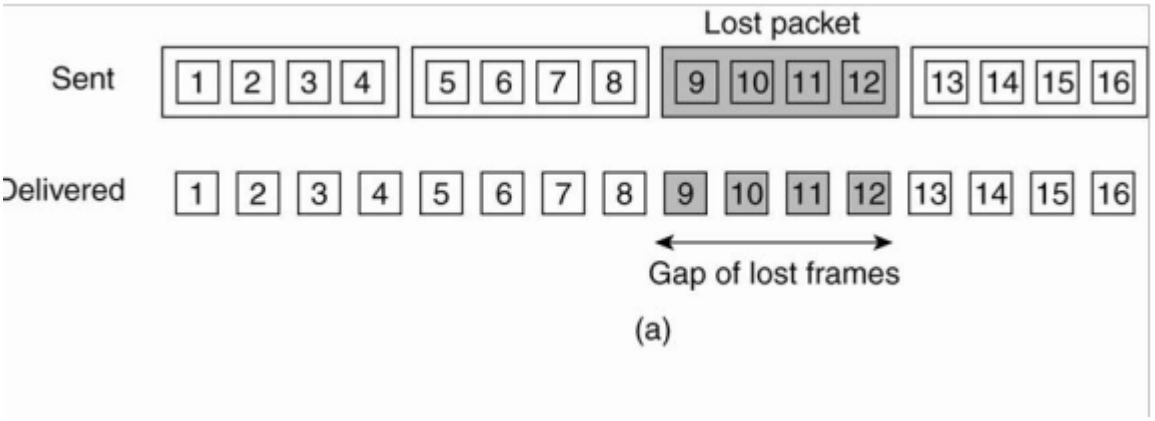
IP is a best effort protocol! Wasn't design to stream data. IP has a little bit of QoS called Differentiated Services field(TOS, has about 6 bit for the code and 2 bit for the explicit congestion notification). Stream of data works because the network has an overcapacity.

Enforcing QoS at the application layer

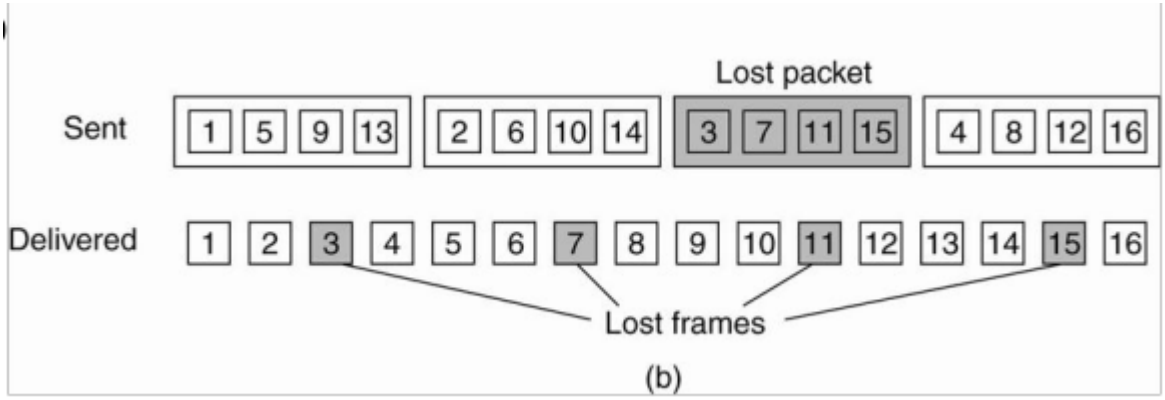
Buffering:Control max jitter by sacrificing session setup time. Tries to resolve the variation of speed in the network creating buffers. Usually the application accumulate some packages to support the times when there are some delay in the network without stuttering



Forward error correction: if you reach a wrong state you tries to jump to the correct state. In the packet you need to have additional information to permit this type of jumps.



Interleaving data: to mitigate the impact of lost packets we separate the message in various packet in a way you have consecutive frames in different packets to permit an easier reconstruction if a packet is lost



Stream Synchroinization

Either you put together the part in the streams and so you only have to handle a single stream at the arrival node or you use the various streams in the network but you have to recompose the initial stream at the arrival node.