PETRA CHRISTIAN UNIVERSITY

Distributed Systems

Communication

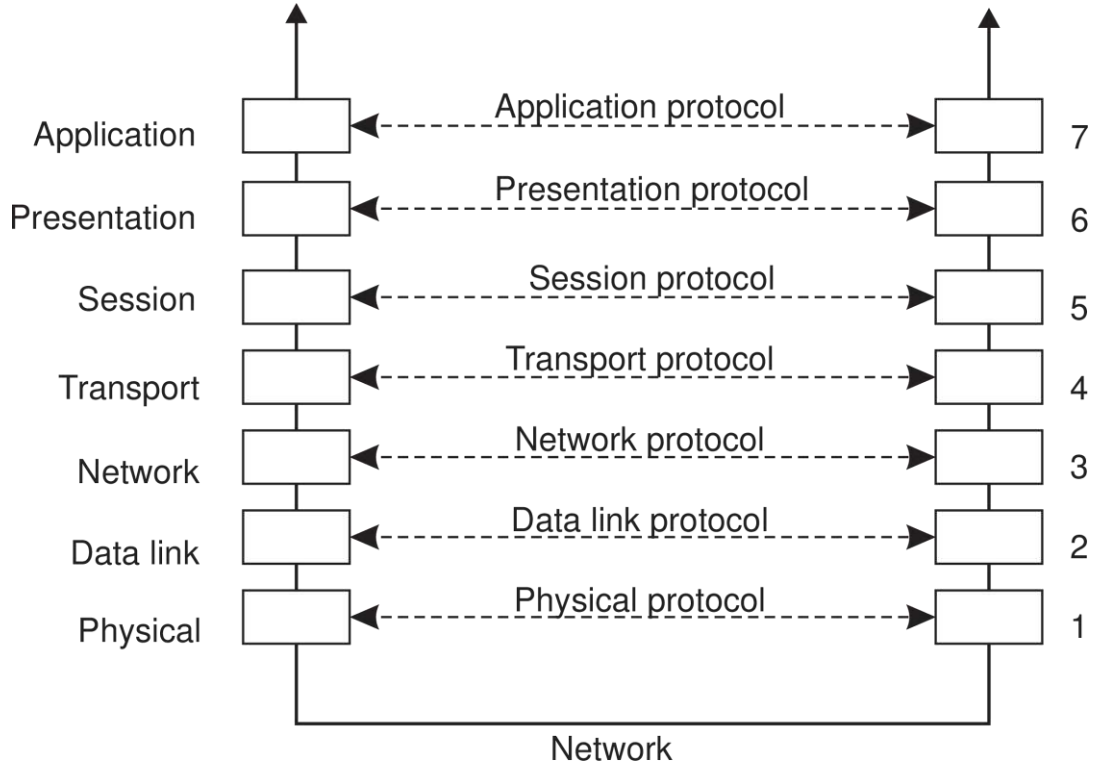
Lecture 04

Fundamentals *(1)*

- Interprocess communication is at the heart of all DSes
- Due to the absence of shared memory, all communication in DSes is based on sending and receiving (low level) messages
- Communication through message passing is harder than using primitives based on shared memory
  - Modern DSes often consist of thousands or even millions of processes scattered across an unreliable network
  - Many different agreements (e.g., protocols) are needed between the communicating parties, varying from the low-level details of bit transmission to the high-level details of how information is to be expressed

Fundamentals *(2)*

- Basic networking model



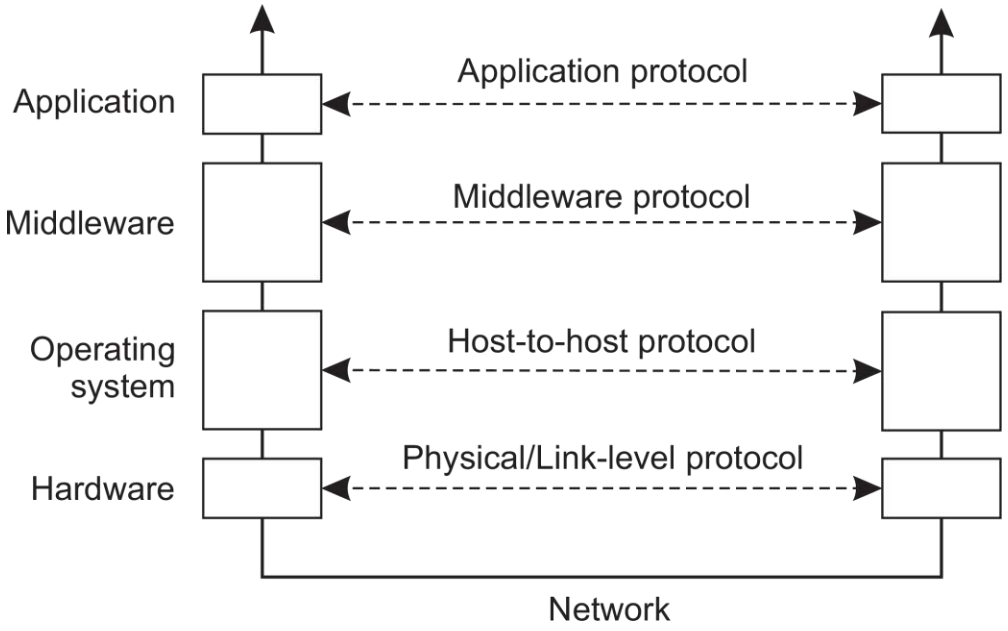
  - For many DSes, the lowest-level interface is that of the network layer

Fundamentals *(3)*

- Transport Layer: provides the actual communication facilities for most DSes
  - TCP: connection-oriented, reliable, stream-oriented communication
  - UDP: unreliable (best-effort) datagram communication
  - RTP (Real-time Transport Protocol): specifies packet formats for real-time data without providing the actual mechanisms for guaranteeing data delivery, and specifies a protocol for monitoring and controlling data transfer
  - SCTP (Streaming Control Transmission Protocol): groups data into messages (i.e., an alternative to TCP that merely moves bytes between processes)

Fundamentals *(4)*

- Middleware protocols: provide common services and protocols that can be used by many different apps



Adapted reference model for networked communication

- A rich set of communication protocols
- (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 & caching

Fundamentals *(5)*

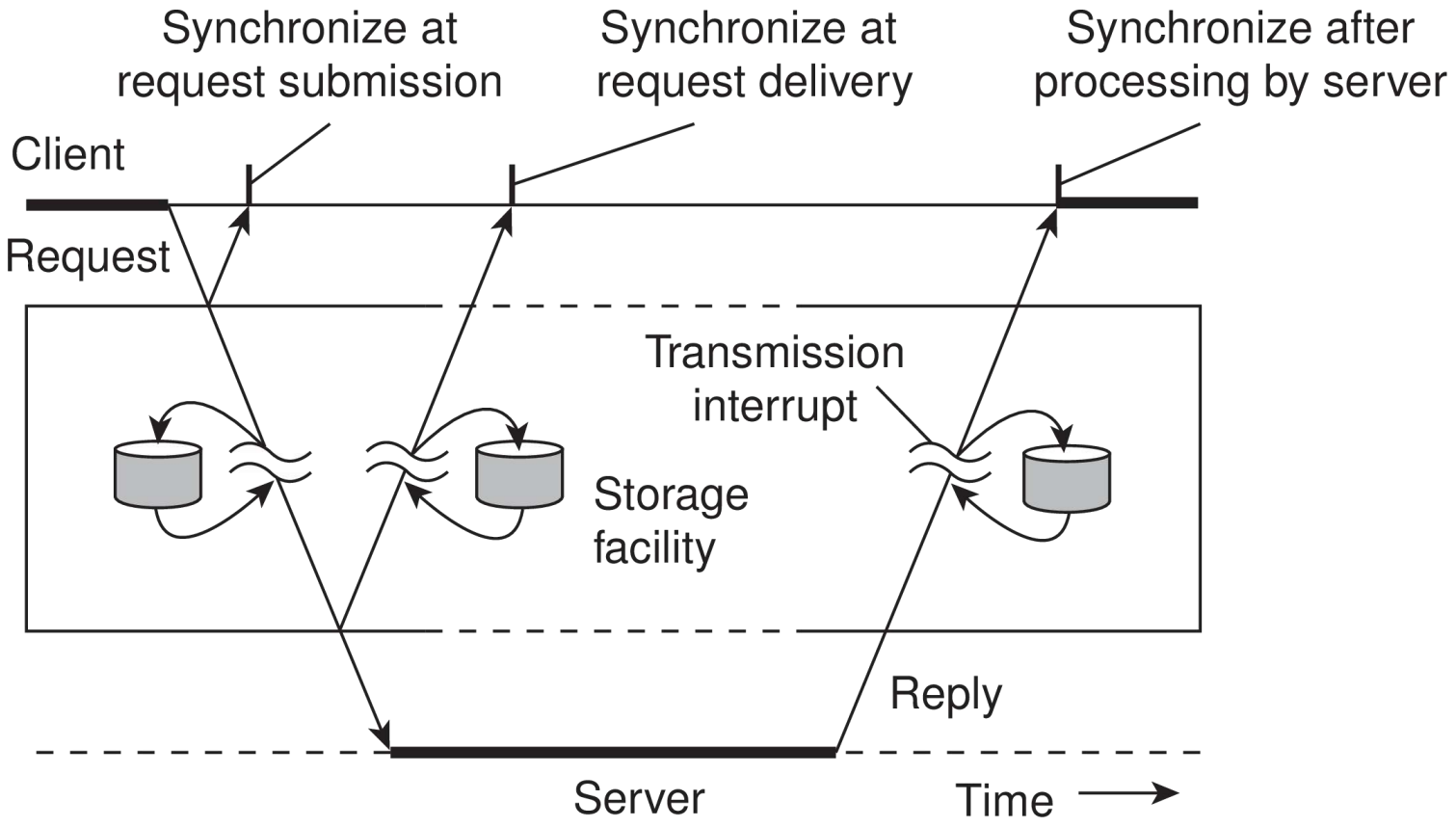
- Types of communication
  - Persistent communication → the message that has been submitted for transmission is stored by the communication middleware as long as it takes to deliver it to the receiver
    - The sending application need not continue execution after submitting the message
    - The receiving application need not be executing when the message is submitted
  - Transient communication → the message is stored by the communication system only as long as the sending and receiving applications are executing
    - The communication system consists of traditional store-and-forward routers
    - If it cannot deliver a message, it will simply drop the message

Fundamentals *(6)*

- Types of communication (cont'd)
  - Asynchronous communication → the sender continues immediately after it has submitted its message for transmission
  - Synchronous communication → the sender is blocked until its request is known to be accepted
    - The sender may be blocked until the middleware notifies that it will take over transmission of the request
    - The sender may synchronize until its request has been delivered to the intended recipient
    - The sender may wait until its request has been fully processed, i.e., up to the time that the recipient returns a response
  - Various combinations of persistence and synchronization occur in practice, e.g., message-queuing systems, RPCs

Fundamentals *(7)*

- Types of communication (cont'd)



Fundamentals *(8)*

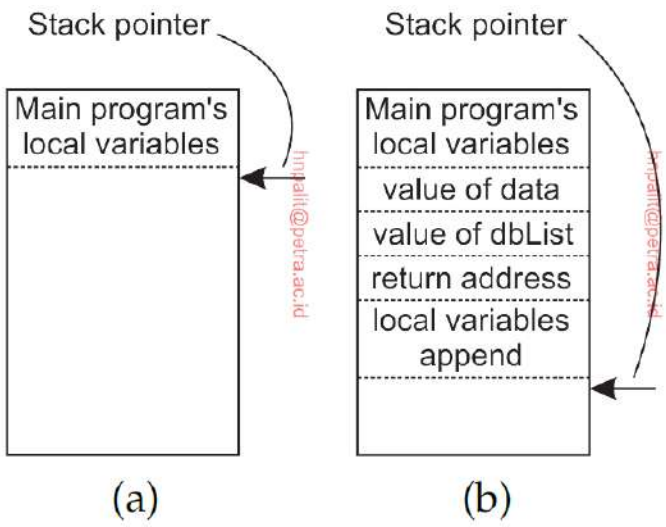
- Types of communication (cont'd)
  - Discrete communication → the parties communicate by messages, each of which forms a complete unit of information
  - Streaming communication → sending multiple messages, one after another, where the messages are related to each other by the order they are sent, or because there is a temporal relationship

Remote Procedure Call *(1)*

- Many DSes have been based on explicit message exchange between processes; however, the **`send`** and **`receive`** do not conceal communication at all (i.e., no access transparency)
- Birrell and Nelson [1984] suggested to allow programs to call procedures located on other machines; this method is known as Remote Procedure Call (RPC)
- The basic idea sounds simple and elegant, but subtle problems exist
  - They execute in different address space, and parameters & results have to be passed, which can be complicated if the machines are not identical
  - Either or both machines can crash & cause more problems

Remote Procedure Call *(2)*

- Conventional procedure call
  - Consider operation **`newlist = append(data, dbList);`**
  - The purpose is to take a *globally* defined list object (i.e., **`dbList`**) and append a single data element to it (i.e., **`data`**)
  - In various programming languages, **`dbList`** is a reference to a list object, whereas **`data`** may be represented by its value
  - When calling **`append`**, both the representations of **`data`** and **`dbList`** are pushed onto the stack, making them accessible to the implementation of **`append`**
  - Variable **`data`** follows a *call-by-value* policy and variable **`dbList`** a *call-by-reference*

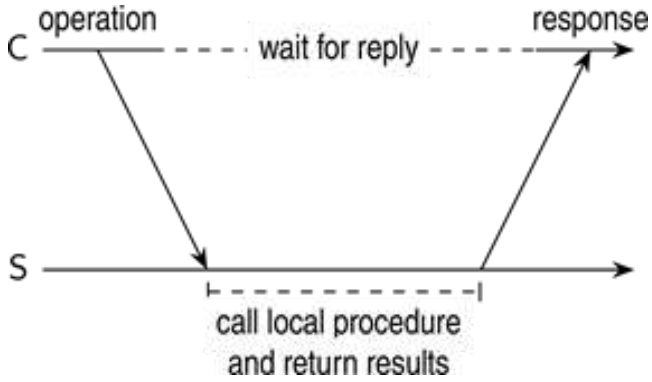


Remote Procedure Call *(3)*

- Conventional procedure call (cont'd)
  - A value parameter is just an initialized local variable; the called procedure may modify it, but do not affect the original at the calling side
  - A reference parameter is a pointer to a variable, so the address of the variable as stored in main memory is pushed onto the stack; when a call to **`append`** adds the value of **`data`** to **`dbList`**, the list object in main memory is modified
  - The difference between call-by-value and call-by-reference is quite important for RPC
  - The decision of which parameter passing mechanism to use is made by the language designers & a fixed property of the language

Remote Procedure Call *(4)*

- Client and server stubs
  - The idea behind RPC is to make it look as much as possible like a local one; i.e., make it transparent – the calling procedure should not be aware that the called procedure is executing on a different machine or vice versa
  - When **`append`** is actually a remote procedure, a different version of **`append`** – called a client stub – is offered
    - It is called using the normal calling sequence like the original one
    - It does not perform an **`append`** operation; instead, it packs the parameters into a message and requests that message to be sent to the remote server



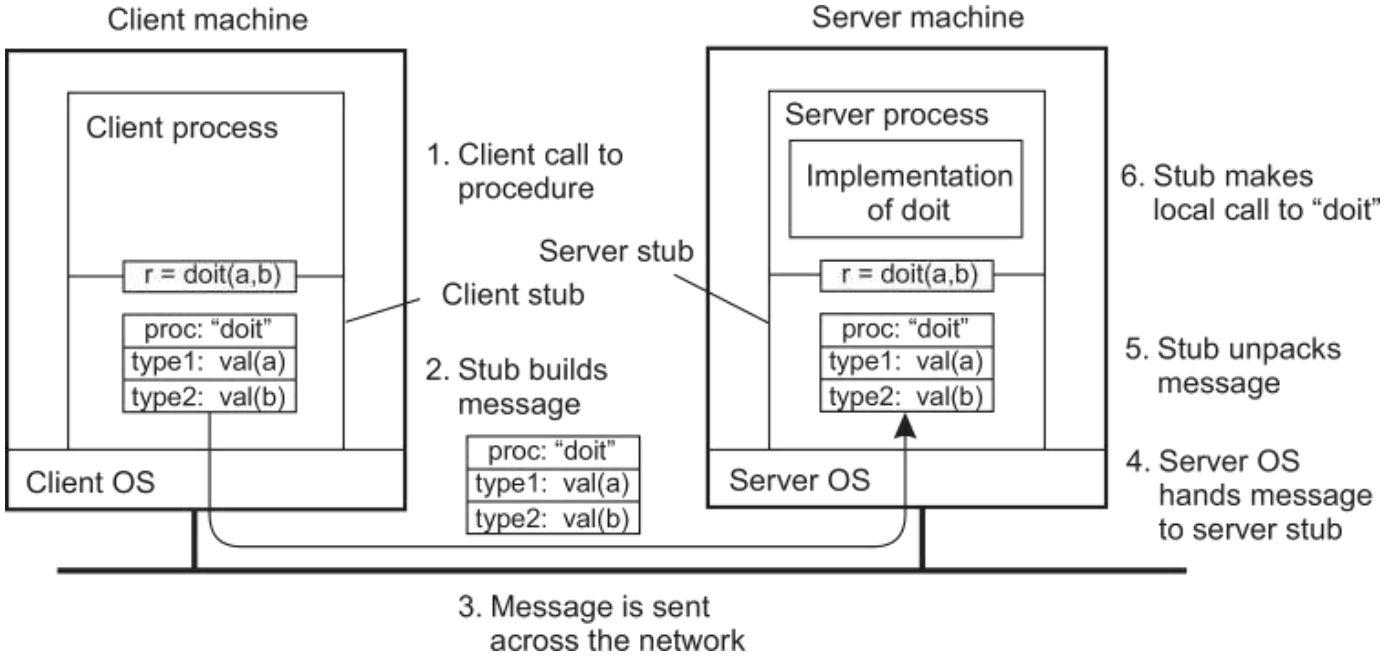
    - Following the call to **`send`**, the client stub calls **`receive`**, blocking itself until the reply comes back

Remote Procedure Call *(5)*

- Client and server stubs (cont'd)
  - When the message arrives at the server, the server's OS passes it up to a server stub (i.e., the server-side equivalent of a client stub)
    - Typically the server stub will have called **`receive`** and be blocked waiting for incoming messages
    - It transforms requests coming in over the network into local procedure calls
    - The server performs its work and returns the result to the caller (i.e., server stub)
    - When the server stub gets control back, it packs the result in a message and calls **`send`** to return it to the client
    - After that, the server stub usually does a call to **`receive`** again
  - When the message arrives at the client machine, the OS passes it to the client stub, which returns it to the caller

Remote Procedure Call *(6)*

- Client and server stubs (cont'd)



1. Client procedure calls client stub.
2. Stub builds message; calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to stub.
5. Stub unpacks parameters; calls server.
6. Server does local call; returns result to stub.
7. Stub builds message; calls OS.
8. OS sends message to client's OS.
9. Client's OS gives message to stub.
10. Client stub unpacks result; returns to client.

Remote Procedure Call *(7)*

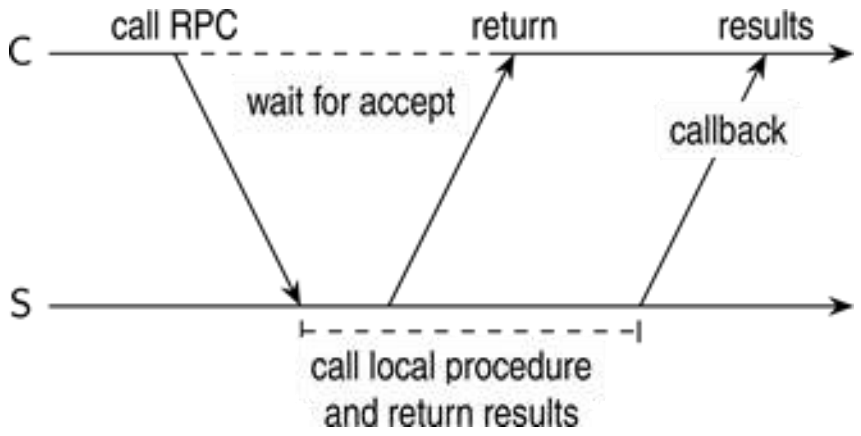
- Parameter passing
  - Packing parameters into a message is called *parameter marshaling*
  - There is more than just wrapping parameters into a message
    - Client and server machines may have different data representations (i.e., byte ordering – little vs. big endian)
    - Wrapping a parameter means transforming a value into a sequence of bytes
    - Client and server have to agree on the same encoding:
      - How are basic data values represented (integers, floats, characters)
      - How are complex data values represented (arrays, unions)
      - ☞ Both need to properly interpret messages → transforming them into machine-independent representations

Remote Procedure Call *(8)*

- Parameter passing (cont'd)
  - How are pointers, or in general, references passed?
    - One solution is to forbid pointers & reference params in general
    - Another strategy is to copy the array (according to its size) into the message and send it to the server; in effect, call-by-reference has been replaced by call-by-copy/restore
    - When the client stub knows the referred data will be only read, there is no need to copy it back when the call has finished
    - Using global references (i.e., meaningful to the calling and the called processes); e.g., if the client and the server have access to the same file system, a file handle (instead of a pointer) is passed

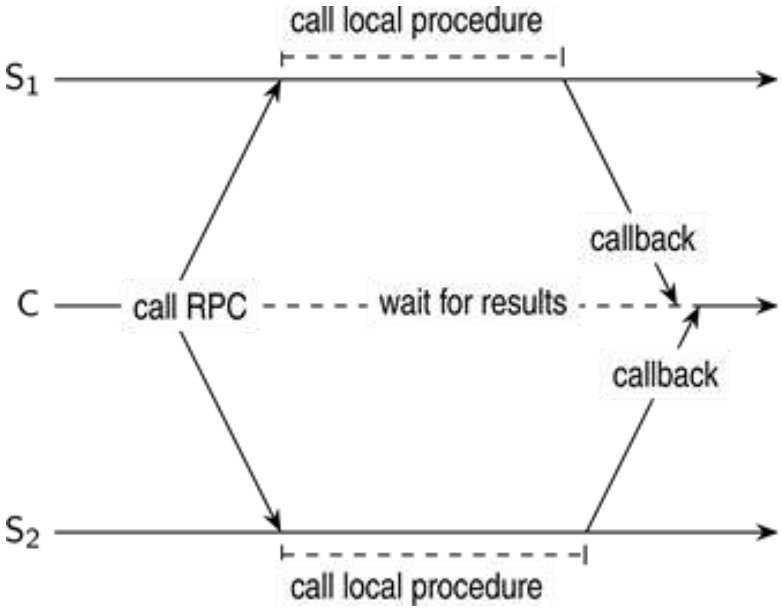
Remote Procedure Call *(9)*

- Variations on RPC: Asynchronous RPC
  - The server immediately sends a reply back to the client the moment the RPC request is received; the reply acts as an acknowledgement to the client
  - The client will continue without further blocking as soon as it has received the server's acknowledgement
  - Deferred synchronous RPC → combining an async RPC with a callback
  - One-way RPC → client does not wait for an ack



Remote Procedure Call *(10)*

- Variations on RPC: Multicast RPC
  - Executing multiple RPCs at the same time → adopting the one-way RPCs to send an RPC request to a group of servers
  - Issues to consider:
    - The client app may be unaware that an RPC is actually being forwarded to multiple servers (e.g., to increase fault tolerance, have all operations executed by a backup server who can take over when the main server fails)
    - Will the client proceed after all responses have been received, or wait just for one? It depends whether the server has been replicated for fault tolerance or to do the same work but on different parts of the input



Message-Oriented Communication

- RPC and remote object invocations are not always appropriate, particularly when it cannot be assumed that the receiving side is executing at the time a request is issued
- *Messaging* or *message-oriented communication* is an alternative communication service to RPCs

- Two types of message-oriented communication:
  - Message-oriented transient communication
  - Message-oriented persistent communication

Msg-Oriented Transient Comm. *(1)*

- Many distributed systems and applications are built directly on top of the simple message-oriented model offered by the transport layer
- Standard interfaces have been introduced to the transport layer to allow programmers to make use of its entire suite of (messaging) protocols through a simple set of operations
- The standard interfaces also make it easier to port an application to a different machine
- An example is the sockets interface introduced in the 1970s in Berkeley UNIX, known as *Berkeley Sockets*

Msg-Oriented Transient Comm. *(2)*

- Berkeley sockets
  - Conceptually, a socket is a communication end point to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read
  - A socket forms an abstraction over the actual port that is used by the local OS for a specific transport protocol (e.g., TCP)
  - Servers generally execute the first four operations (i.e., *socket*, *bind*, *listen*, and *accept*), normally in the order given

Msg-Oriented Transient Comm. *(3)*

- Berkeley sockets (cont'd)
  - When calling the *socket* operation, the caller creates a new comm. end point for a specific transport protocol
    - Internally, it means that the local OS reserves resources to accommodate sending and receiving messages for the specified protocol
  - The *bind* operation associates a local address with the newly-created socket, e.g., a server should bind the IP address of its machine together with a (possibly well-known) port number to a socket
    - Binding tells the OS that the server wants to receive messages only on the specified address and port

Msg-Oriented Transient Comm. *(4)*

- Berkeley sockets (cont'd)
  - The *listen* operation is called only in the case of connection-oriented communication
    - It is a non-blocking call that allows the local OS to reserve enough buffers for a specified maximum number of pending connection requests that the caller is willing to accept
  - A call to *accept* blocks the caller until a connection request arrives
    - When a request arrives, the local OS creates a new socket with the same properties as the original one, and returns it to the caller
    - This approach will allow the server to, for example, fork a process that will subsequently handle the actual comm. through the new connection; the server, in the meantime, can go back and wait for another connection request on the original socket

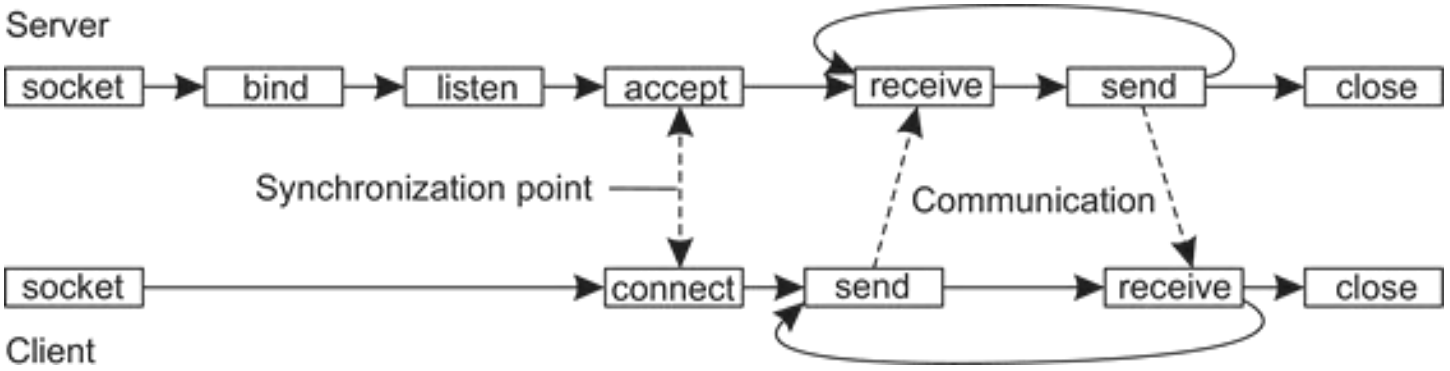
Msg-Oriented Transient Comm. *(5)*

- Berkeley sockets (cont'd)
  - At the client side, the order of execution is as the following
    - A socket must first be created using the *socket* operation, but explicitly *binding* the socket to a local address is not necessary, since the OS can dynamically allocate a port when the connection is set up
    - The *connect* operation requires that the caller specifies the transport-level address to which a connection request is to be sent; the client is blocked until a connection has been set up successfully
    - After that, both sides can start exchanging information through the *send* and *receive* operations
    - Finally, closing a connection is symmetric when using sockets, and is established by having both the client and server call the *close* operation

Msg-Oriented Transient Comm. *(6)*

- Berkeley sockets (cont'd)

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



Msg-Oriented Transient Comm. *(7)*

- Sockets are rather low level and programming mistakes are easily made
- More advanced approaches for msg-oriented comm. is needed to:
  - make network programming easier
  - expand beyond the functionality offered by existing networking protocols
  - make better use of local resources
  - etc.

Msg-Oriented Transient Comm. *(8)*

- ZeroMQ
  - Provides a higher level of expression by pairing sockets: one for sending messages and a corresponding one for receiving messages
  - Supports *many-to-one* and *one-to-many* communication
  - All communication is asynchronous
  - Three communication patterns supported:
    - *Request-reply* – used in traditional client-server communication
    - *Publish-subscribe* – clients subscribe to specific messages that are published by servers
    - *Pipeline* – a process wants to push out its results, assuming that there are other processes that want to pull in those results

Msg-Oriented Transient Comm. *(9)*

- Sockets were deemed insufficient for communication in high-performance multicomputers:
  - They were at the wrong level of abstraction by supporting only simple **`send`** and **`receive`** operations
  - Sockets had been designed to communicate across networks using general-purpose protocol stacks (e.g., TCP/IP), not suitable for the proprietary protocols for high-speed interconnection networks
- The need to be hardware and software independent eventually led to the definition of a standard called the *Message-Passing Interface (MPI)*
  - Designed for parallel apps and tailored to transient comm.
  - Make direct use of the underlying network
  - Assume that serious failures do not require auto recovery

Msg-Oriented Transient Comm. *(10)*

- Message-Passing Interface
  - MPI assumes that comm. takes place within a known group of processes; each group is assigned an identifier, and each process within a group is also assigned a (local) id
  - A (*groupID*, *processID*) pair uniquely identifies the source or destination of a message, and is used instead of a transport-level address
  - There may be several, overlapping groups of processes involved in a computation and that are all executing at the same time
  - At the core of MPI are messaging operations to support transient comm., of which the most intuitive ones are discussed in the following slides

Msg-Oriented Transient Comm. *(11)*

- Message-Passing Interface (cont'd)
  - The **`MPI_SEND`** operation – which is implementation dependent – is a blocking send operation that may block the caller either until the specified message has been copied to the MPI runtime system at the sender's side, or until the receiver has initiated a receive operation
  - The **`MPI_BSEND`** operation supports transient async comm.
    - The sender submits a message for transmission, which is generally first copied to a local buffer in the MPI runtime system
    - When the message has been copied, the sender continues
    - The local MPI runtime system will remove the message from its local buffer and take care of transmission as soon as a receiver has called a receive primitive
  - The **`MPI_SSEND`** operation is sync comm. by which the sender blocks until its request is accepted for further processing

Msg-Oriented Transient Comm. *(12)*

- Message-Passing Interface (cont'd)
  - The **`MPI_SENDRECV`** operation gives the strongest form of sync comm., in which it sends a request to the receiver and blocks until the latter returns a reply; basically, it corresponds to a normal RPC
  - The **`MPI_ISEND`** operation is a variant of **`MPI_SEND`** that supports async comm., in which the sender passes a pointer to the message (i.e., avoiding copying messages from user buffers to buffers internal to the local MPI runtime system) and immediately continues
  - Likewise, the **`MPI_ISSEND`** operation is the async variant of **`MPI_SSEND`**, in which the sender passes only a message's pointer to the MPI runtime system and continues after the runtime system indicates it has processed the message

Msg-Oriented Transient Comm. *(13)*

- Message-Passing Interface (cont'd)
  - The **`MPI_RECV`** operation is called to receive a message; it blocks the caller until a message arrives
  - The **`MPI_IRECV`** operation is the async variant, by which a receiver indicates that it is prepared to accept a message

Operation	Description
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 transmission 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

Msg-Oriented Persistent Comm. *(1)*

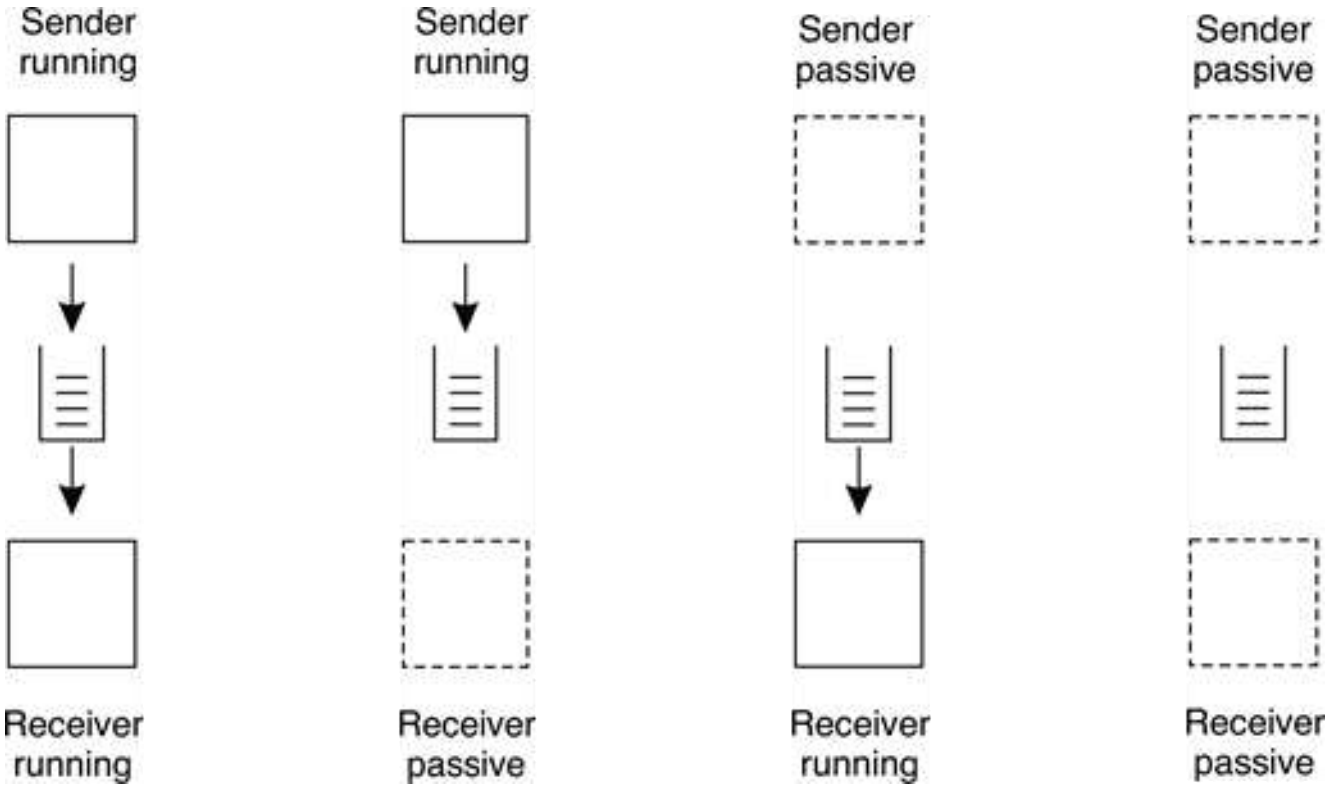
- Message-queuing systems, often called Message-Oriented Middleware (MOM), provide extensive support for persistent async comm.
  - They offer intermediate-term storage capacity for messages, without requiring either the sender or receiver to be active during message transmission
  - They are typically targeted to support message transfers that are allowed to take minutes instead of seconds or milliseconds
- The basic idea is that applications communicate by inserting messages in specific queues
- The messages are forwarded over a series of comm. servers and are eventually delivered to the destination (which could be down)

Msg-Oriented Persistent Comm. *(2)*

- In practice, most comm. servers are directly connected to each other; so, a message is generally transferred directly to a destination server
- In principle, each application has its own private queue to which other applications can send messages; the queue can be read only by its associated application, but it is also possible for multiple applications to share a single queue
- A sender is generally given only the guarantees that its message will eventually be inserted in the recipient's queue; no guarantees about when, or even if the message will actually be read
- The sender & receiver can execute independently

Msg-Oriented Persistent Comm. *(3)*

- Four combinations for loosely coupled communication



Msg-Oriented Persistent Comm. *(4)*

- The only important aspect from the perspective of middleware is that messages are properly addressed
  - Addressing is done by providing a system-wide unique name of the destination queue
  - Message size may be limited in some cases, although it is also possible that the underlying system takes care of fragmenting and assembling large messages in a way that is completely transparent to applications
- The **`PUT`** operation is a nonblocking call called by a sender to pass a message to the underlying system that is to be appended to the specified queue
- The **`GET`** operation is a blocking call by which an authorized process can remove the longest pending message in the specified queue

Msg-Oriented Persistent Comm. *(5)*

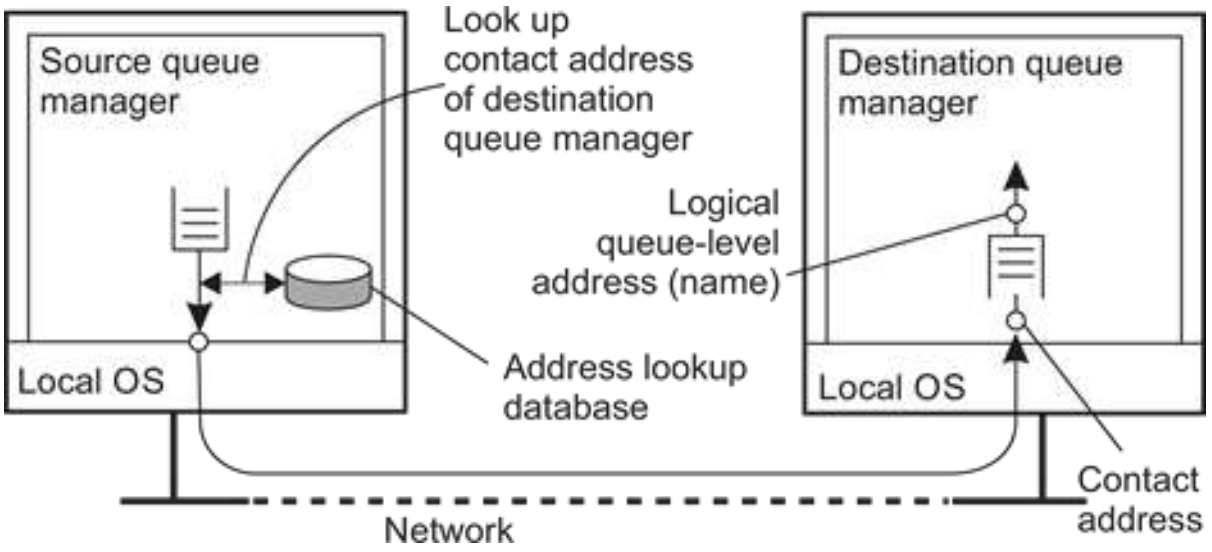
- Variations of the **`GET`** call allow searching for a specific message in the queue, e.g., using a priority or a matching pattern
- The nonblocking variant is given by the **`POLL`** operation, which simply continues if the queue is empty or if a specific message could not be found
- Most queuing systems also allow a process to install a handler as a *callback function* (through a **`NOTIFY`** operation), automatically invoked whenever a message is put into the queue
  - Callbacks can also be used to automatically start a process – if none is executing – that will fetch messages from the queue
  - Often implemented by means of a daemon that monitors the queue for incoming messages and handles accordingly

Msg-Oriented Persistent Comm. *(6)*

Operation	Description
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

- It is the responsibility of a message-queuing system to provide queues to senders & receivers and take care that messages are transferred from their source to their destination queue
- The collection of queues is distributed across multiple machines; thus, a message-queuing system should maintain a (possibly distributed) database that maps queue names to network locations

Msg-Oriented Persistent Comm. *(7)*



- Queues are managed by queue managers
  - Normally, a queue manager interacts directly with the application that is sending or receiving a message
  - Some special queue managers operate as *routers* or *relays* as they forward incoming messages to other managers
  - A message-queuing system may gradually grow into a complete, application-level, *overlay network*

Msg-Oriented Persistent Comm. *(8)*

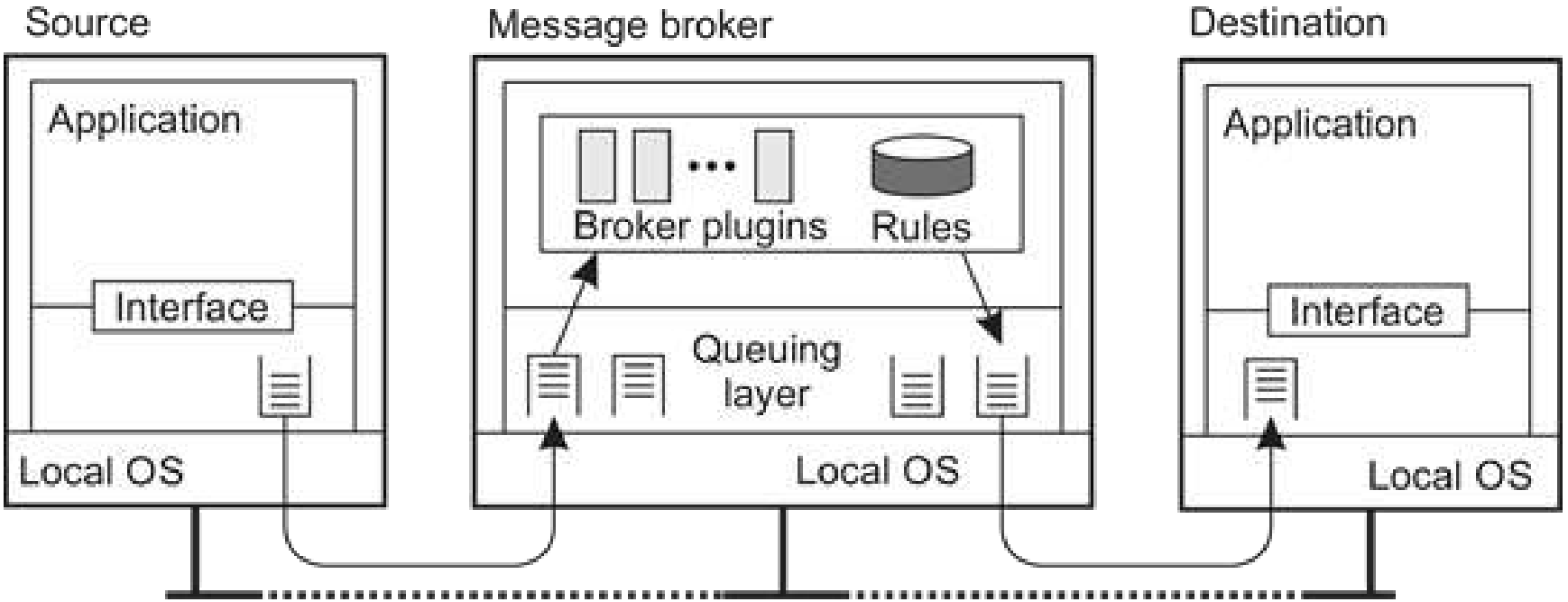
- An important application area of message-queuing systems is integrating existing and new applications into a single, coherent distributed system
  - It requires that applications can understand the messages they receive; it requires the sender to have its outgoing messages in the same format as that of the receiver
  - The problem is that each time an application requiring a separate messaging protocol is added to the system, other application communicating with it will need to provide the means for converting their respective messages
- An alternative is to agree on a common messaging protocol; however, it makes sense only if the collection of processes that make use of that protocol indeed have enough in common

Msg-Oriented Persistent Comm. *(9)*

- The general approach is to learn to live with differences, and try to provide the means to make conversions as simple as possible
  - Conversions are handled by *message brokers*
  - A message broker acts as an application-level gateway in a message-queuing system; a message broker is generally not considered an integral part of the queuing system
  - It can be as simple as a reformatter for messages, e.g., changing record delimiters and field formats
  - It may handle conversion between two different database applications
  - More common is its use for advanced *enterprise application integration* (EAI), like matching apps based on the messages being exchanged (i.e., pub-sub model)

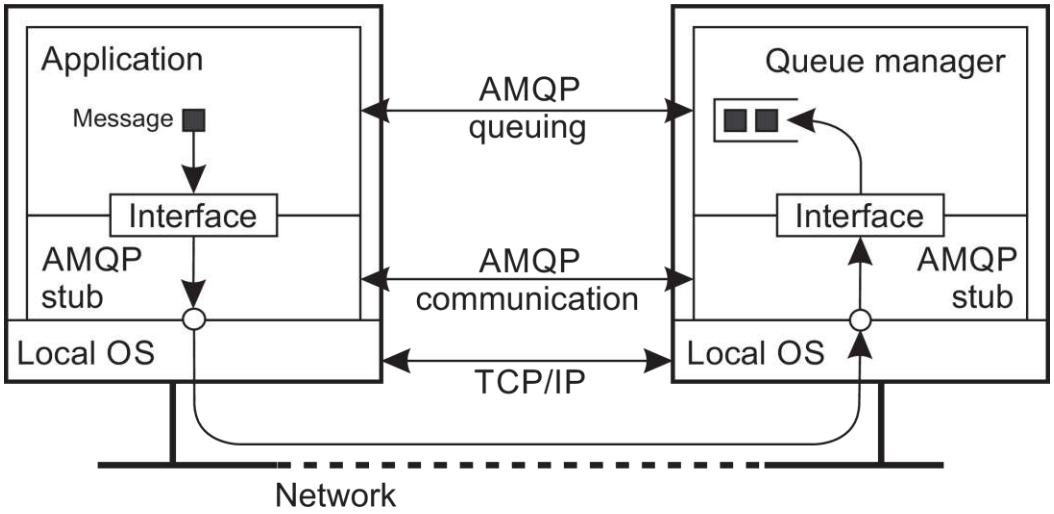
Msg-Oriented Persistent Comm. *(10)*

- At the heart of a message broker lies a repository of rules for transforming a message of one type to another; the problem is defining the rules and developing the plugins



Example: AMQP *(1)*

- Advanced Message-Queuing Protocol was intended to play the same role as, e.g., TCP in networks: a protocol for high-level messaging with different implementations



- Implementations of AMQP: RabbitMQ & Apache's ActiveMQ
- AMQP revolves around applications, queue managers, and queues

- Basic model:
  - App sets up a *connection* (i.e., a container for a number of *one-way channels*) to a queue manager; two one-way channels form a *session*
  - A *link* is akin to a socket and maintains state about message transfers

Example: AMQP *(2)*

- When a message is to be transferred, the app passes it to local AMQP stub; message transfer normally proceeds in 3 steps:
  - At the sender's side, the message is assigned a unique ID and is recorded locally to be in an *unsettled state*. The stub subsequently transfers the message to the server, where the AMQP stub also records it as being in an unsettled state. Then, the server-side stub passes it to the queue manager.
  - The receiving app (i.e., queue manager) is assumed to handle the message and normally reports back to its stub that it is finished. The stub passes this info to the original sender, at which point the message at the original sender's AMQP stub enters a *settled state*.
  - The AMQP stub of the original sender now tells the stub of the original receiver that message transfer has been settled (i.e., the original sender will forget about the message from now on). The receiver's stub can now also discard anything about the message, formally recording it as being settled as well.

Multicast Communication *(1)*

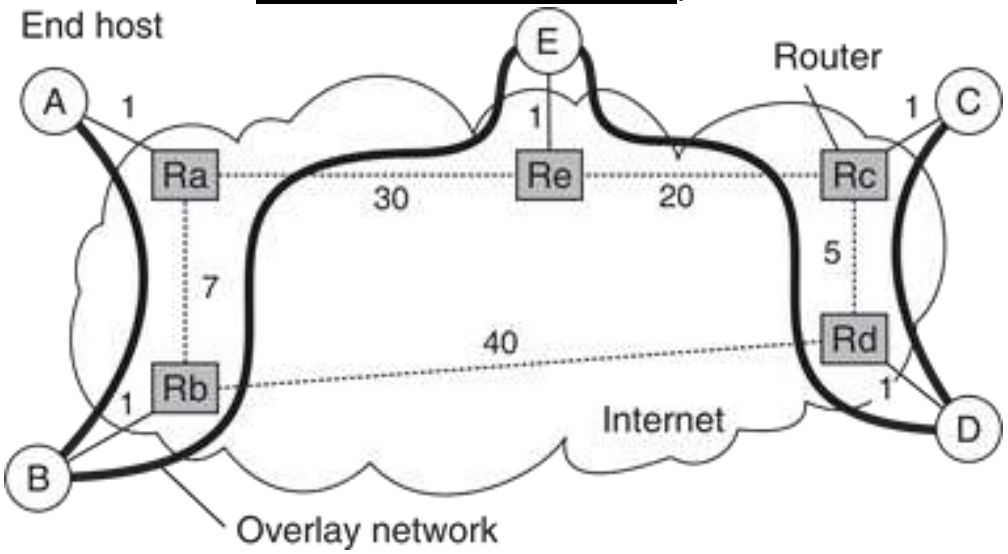
- Multicast communication in DSes is the support for sending data to multiple receivers
  - For many years, this topic has belonged to the domain of network protocols; some are network-level solutions and the others are transport-level solutions
  - A major issue in all solutions is setting up the comm. paths for information dissemination; a huge management effort is involved, human intervention is required in many cases
- With the advent of P2P technology, and notably structured overlay management, it becomes easier to set up comm. paths; as P2P solutions are typically deployed at the application layer, various application-level multicasting techniques have been introduced

Multicast Communication *(2)*

- Application-level multicasting
  - Basic idea: nodes are organized into an overlay network, which is then used to disseminate info to its members
  - Network routers are not involved in group membership; consequently, the connections between nodes in the overlay network may cross several physical links, and as such, routing messages may not be optimal
  - Two approaches in the construction of the overlay network
    - Nodes may organize themselves directly into a tree, meaning that there is a unique (overlay) path between every pair of nodes
    - Nodes may organize into a mesh network, in which every node will have multiple neighbors and, in general, there exist multiple paths between every pair of nodes
    - ☞ The latter provides higher robustness

Multicast Communication *(3)*

- Application-level multicasting (cont'd)
  - Building a tree is not difficult once we have organized the nodes into an overlay, but building an efficient tree may be a different story
  - The figure shows a set of five nodes organized in a simple overlay network; node *A* is the root of a multicast tree



- Whenever *A* multicasts a msg to the other nodes, the msg will traverse links <*B*, *Rb*>, <*Ra*, *Rb*>, <*E*, *Re*>, <*Rc*, *Rd*>, <*D*, *Rd*> twice
- The overlay network would have been more efficient if we had not constructed overlay links <*B*, *E*> & <*D*, *E*>, but instead <*A*, *E*> & <*C*, *E*>

Multicast Communication *(4)*

- Application-level multicasting (cont'd)
  - The quality of an application-level multicast tree is generally measured by three different metrics:
    - *Link stress* → counts how often a packet crosses the same link (when it is greater than 1, it means a packet may be forwarded along two different connections at a logical level, but part of those connections may actually correspond to the same physical link)
    - *Stretch or Relative Delay Penalty (RDP)* → measures the ratio of the delay between two nodes in the overlay against the delay that those two nodes would experience in the underlying network (when constructing an overlay network, the goal is to minimize the aggregated stretch or the avg RDP measured over all node pairs)
    - *Tree cost* → a global metric related to minimizing the aggregated link costs (e.g., if the cost is taken to be the delay, then optimizing the tree cost boils down to finding a minimal spanning tree in which the total time for disseminating info to all nodes is minimal)

Multicast Communication *(5)*

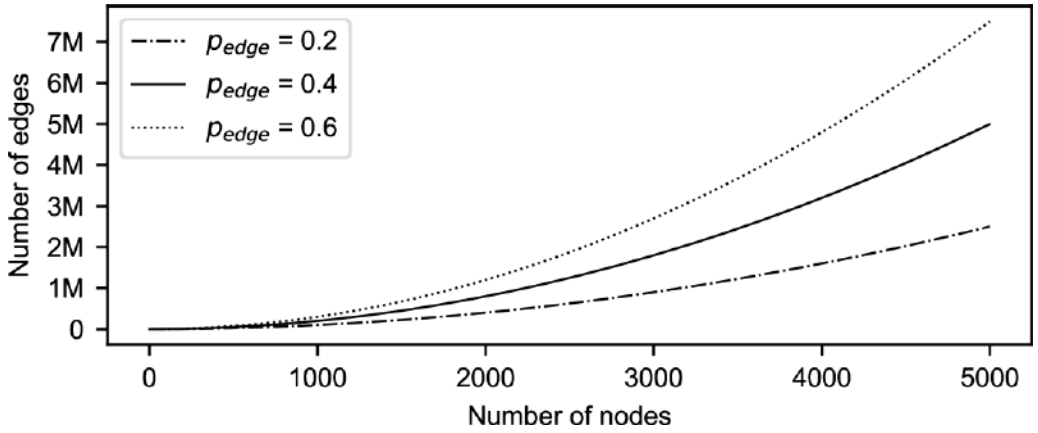
- Application-level multicasting (cont'd)
  - Switch-trees solution
    - Assume we have a multicast tree with a single source as root
    - In this tree, a node *P* can switch parents by dropping the link to its current parent in favor of a link to another node; constraints are:
      - The new parent can never be a member of the subtree rooted at *P* (as this would partition the tree and create a loop)
      - The new parent will not have too many immediate children (to limit the load of forwarding messages by any single node)
    - Different criteria for deciding to switch parents:
      - Optimizing the route to the source; to this end, each node regularly receives info on other nodes so it can evaluate whether another node would be a better parent
      - The delay to the potential other parent is lower than to the current parent; this simple scheme is a reasonable heuristic leading to a good approximation of a minimal spanning tree

Multicast Communication *(6)*

- Application-level multicasting (cont'd)
  - Switch-trees solution (cont'd)
    - For an example:
      - Node *P* receives info on the neighbors of its parent; the neighbors consists of *P*'s grandparent and the other siblings of *P*'s parent
      - Node *P* can then evaluate the delays to each of these nodes and choose the one with the lowest delay, say *Q*, as its new parent; to that end, it sends a switch request to *Q*
      - To prevent loops from being formed due to concurrent switching requests, a node that has an outstanding switch request will simply refuse to process any incoming requests; in effect, only completely independent switches can be carried out simultaneously
    - Whenever a node notices that its parent has failed, it simply attaches itself to the root; at that point, the optimization protocol can proceed as usual and will eventually place the node at a good point in the multicast tree

Multicast Communication *(7)*

- Flooding-based multicasting

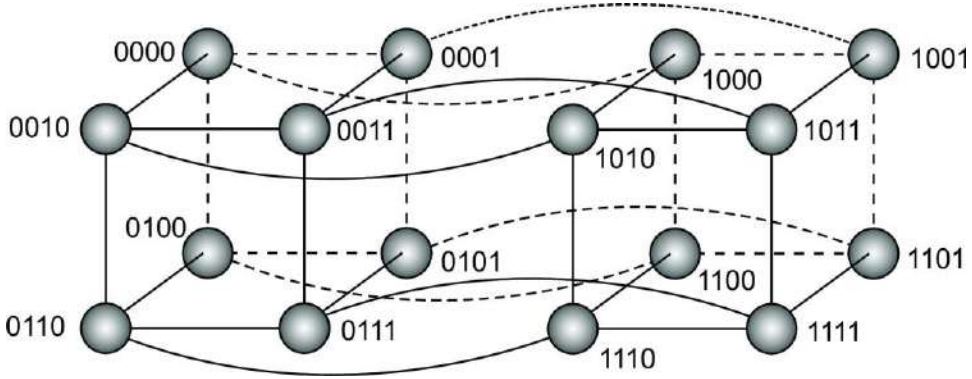


- *P* simply sends a message *m* to each of its neighbors
- Each neighbor will forward that message, except to *P*, and only if it had not seen *m* before

- Variation (probabilistic broadcasting): Let *Q* forward a message with a probability P_{flood}, possibly even dependent on its own number of neighbors (i.e., node degree) or the degree of its neighbors
  - In a random network of 10,000 nodes and P_{edge} = 0.1, we need only set P_{flood} = 0.01 to establish a more than 50-fold reduction in the number of messages sent in comparison to full flooding

Multicast Communication *(8)*

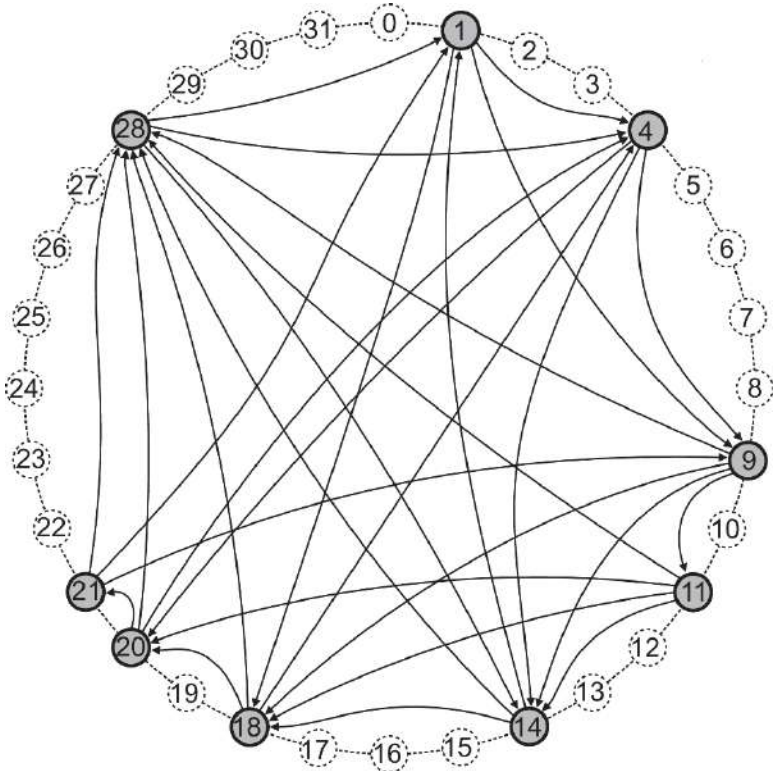
- Flooding-based multicasting (cont'd)
  - When dealing with a structured overlay (i.e., deterministic topology), designing efficient flooding schemes is simpler
    - Consider a 4-dimensional hypercube



- A simple and efficient broadcast scheme relies on keeping track of neighbors *per dimension*
- A node initially broadcasts a message *m* to all of its neighbors and tags *m* with the label (i.e., dimension) of the edge over which it sends the message
- For example: If node 1001 broadcasts a message *m*, it will send (*m*,1) to 0001, (*m*,2) to 1101, (*m*,3) to 1011, (*m*,4) to 1000. When a node receives a broadcast message, it will forward it only along edges with a higher dimension. So, node 1101 will send (*m*,3) to 1111 and (*m*,4) to 1100. It can be shown that every broadcast requires precisely $N-1$ messages (where $N = 2^n$ = the no. of nodes in an n-dim hypercube)

Multicast Communication *(9)*

- Flooding-based multicasting (cont'd)
  - When dealing with a structured overlay … (cont'd)



- Consider a 5-bit Chord
  - Assume that node 9 wants to flood a message to all other nodes
  - Node 9 divides the ID space into four segments (one for each of its neighbors): node 11 takes care of nodes ID $11 \leq k < 14$, node 14 for $14 \leq k < 18$, node 18 for $18 \leq k < 28$, and node 28 for $28 \leq k < 9$
  - Node 28 will subsequently divide the part of the ID space it is requested to handle into two subsegments: [1,4) and [4,9)
  - Node 18 will split its segment into only one part and forward the message to [20,28)
  - Lastly, node 20 forwards the message to [21,28)

Multicast Communication *(10)*

- Gossip-based data dissemination
  - An increasingly important technique for disseminating info is to rely on *epidemic behavior*; epidemics studies the spreading of infectious diseases
  - The main goal is to rapidly propagate info among a large collection of nodes using only local info; there is no central component by which info dissemination is coordinated
  - Using the terminology from epidemics:
    - A node that is part of a DS is called *infected* if it holds data that it is willing to spread to other nodes
    - A node that has not yet seen this data is called *susceptible*
    - An updated node that is not willing or able to spread its data is said to be *removed*

Multicast Communication *(11)*

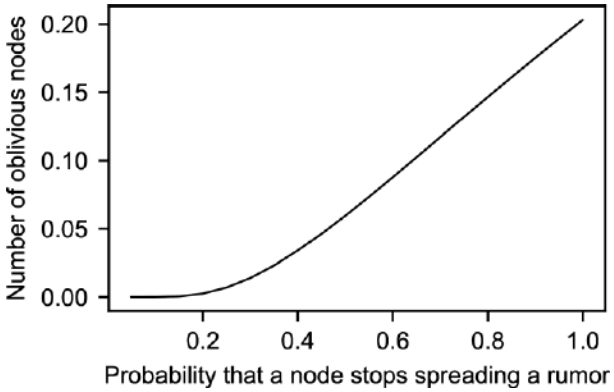
- Gossip-based data dissemination (cont'd)
  - We can distinguish old from new data, e.g., because it has been timestamped or versioned
  - A popular propagation model is that of *anti-entropy*; in this model, a node *P* picks another node *Q* at random, and subsequently exchanges updates with *Q* using one of these three approaches:
    - *P* only pushes its own updates to *Q*
    - *P* only pulls in new updates from *Q*
    - *P* and *Q* send updates to each other (i.e., a push-pull approach)
  - When it comes to rapidly spreading updates, only pushing updates turns out to be a bad choice; if many nodes are infected, the probability of one selecting a susceptible node is relatively small

Multicast Communication *(12)*

- Gossip-based data dissemination (cont'd)
  - The pull-based approach works much better when many nodes are infected; chances are big that a susceptible node will contact an infected one to subsequently pull in the updates and become infected as well
  - If only a single node is infected, updates will rapidly spread across all nodes using either form of anti-entropy, although push-pull remains the best strategy
  - Propagating a single update to all nodes takes $\mathcal{O}(log\ (N))$ rounds, where N is the number of nodes; this indicates that propagating updates is fast, but above all scalable
  - One special variant of this approach is called *rumor spreading*, or simply *gossiping*

Multicast Communication *(13)*

- Gossip-based data dissemination (cont'd)
  - Gossiping
    - If node *P* has just been updated for data item *x*, it contacts an arbitrary other node *Q* and tries to push the update to *Q*
    - However, it is possible that *Q* was already updated by another node; in that case, *P* may lose interest in spreading the update any further (with probability p_{stop}), it then becomes removed
    - It is an excellent way of rapidly spreading news; however, it cannot guarantee that all nodes will actually be updated



  - One of the main advantages of epidemic algorithms is their scalability, due to the fact that the number of sync. between processes is relatively small compared to other propagation methods

Multicast Communication *(14)*

- Gossip-based data dissemination (cont'd)
  - For wide-area systems, it makes sense to take the actual network topology into account to achieve better results
    - Nodes that are connected to only a few other nodes are contacted with a relatively high probability
    - Assumption: such nodes form a bridge to other remote parts of network; therefore, they should be contacted as soon as possible
    - This approach is referred to as *directional gossiping*
  - Most epidemic solutions assume that a node can randomly select any other node to gossip with
    - This implies that, in principle, the complete set of nodes should be known to each member; in a large system, this can never hold
    - Fortunately, maintaining a partial view that is more or less continuously updated will organize the collection of nodes into a random graph

Multicast Communication *(15)*

- Gossip-based data dissemination (cont'd)
  - Epidemic algorithms have a rather strange side-effect: spreading the *deletion* of a data item is hard
    - The problem lies in the fact that deletion of a data item destroys all info on that item
    - Consequently, when a data item is simply removed from a node, the node will eventually receive old copies of the data item and interpret those as updates on something it did not have before
  - The trick is to record the deletion of a data item as just another update and keep a record of that deletion
    - In this way, old copies will not be interpreted as something new, but merely treated as versions that have been updated by a delete operation
    - The recording of a deletion is done by spreading *death certificates*

Multicast Communication *(16)*

- Gossip-based data dissemination (cont'd)
  - The problem with death certificates is that they should eventually be cleaned up, or otherwise each node will gradually build a huge local database of historical info on deleted data items that actually are not used
  - Dormant death certificates
    - Each death cert is timestamped when it is created
    - If it can be assumed that updates propagate to all nodes within a known finite time, then death certs can be removed after this max propagation time has elapsed
    - To provide hard guarantees that deletions are indeed spread to all nodes, only a few nodes maintain dormant death certs that are never thrown away

PETRA
CHRISTIAN
UNIVERSITY

RESERVED MATERIALS

Stream-Oriented Communication *(1)*

- Communication as discussed so far has concentrated on exchanging more-or-less independent and complete units of information
- There are forms of comm. in which timing plays a crucial role, e.g., an audio stream built up as a sequence of 16-bit samples
  - Assume that the audio stream represents CD quality, meaning that the original sound wave has been sampled at a frequency of 44,100 Hz
  - To reproduce the original sound, it is essential that the samples in the audio stream are played out in the order they appear in the stream, but also at intervals of exactly 1 / 44,100 second

Stream-Oriented Communication *(2)*

- Support for the exchange of time-dependent info is often formulated as support for *continuous* media
- In *continuous* (representation) media, the temporal relationships between different data items are fundamental to correctly interpreting what the data actually mean, e.g., an audio or video stream
- In *discrete* (representation) media, the temporal relationships between data items are *not* fundamental to correctly interpreting the data, e.g., text, still images, object code or executable files

Stream-Oriented Communication *(3)*

- To capture the exchange of time-dependent info, DSes generally provide support for data streams
  - A data stream is a sequence of data units
  - It can be applied to discrete as well as continuous media
  - UNIX pipes or TCP/IP connections are typical examples of (byte-oriented) discrete data streams
  - Playing an audio file typically requires setting up a continuous data stream between the file and the audio device
- To capture timing aspects, a distinction is often made between different transmission modes: *asynchronous*, *synchronous*, and *isochronous*

Stream-Oriented Communication *(4)*

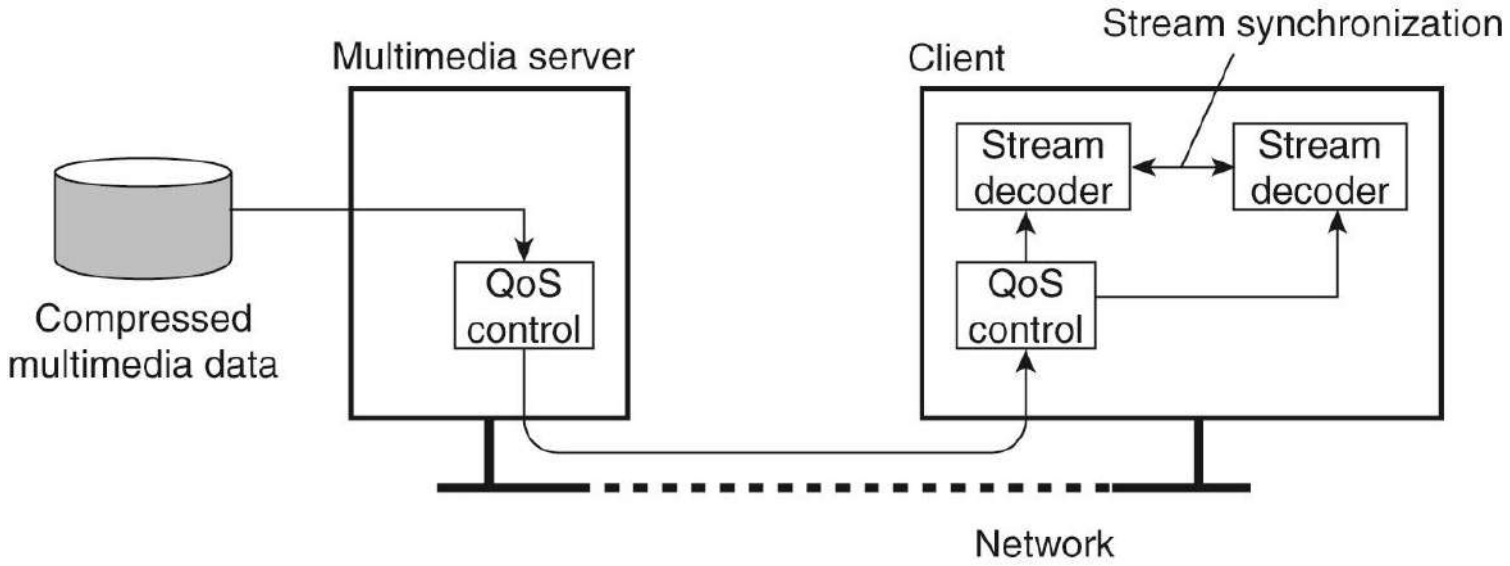
- Asynchronous transmission mode
  - The data items in a stream are transmitted one after the other, but there are no further timing constraints on when transmission of items should take place
  - This is typically the case for discrete data streams
- Synchronous transmission mode
  - There is a maximum end-to-end delay defined for each unit in a data stream
  - It may be important that the end-to-end propagation time through the network is guaranteed to be lower that the time interval between taking samples, but it cannot do any harm if samples are propagated faster than necessary
  - E.g., a sensor sending sample temperature at a certain rate

Stream-Oriented Communication *(5)*

- Isochronous transmission mode
  - It is necessary that data units are transferred on time; this means that data transfer is subject to a maximum and minimum end-to-end delay, also referred to as bounded (delay) jitter
  - It plays a crucial role in representing audio and video
- Streams can be simple or complex
  - A simple stream consists of only a single sequence of data
  - A complex stream consists of several related simple streams, called substreams; the relation between the substreams is often also time dependent

Stream-Oriented Communication *(6)*

- From a DS perspective, we can distinguish several elements needed for supporting streams
  - For simplicity, we concentrate on streaming stored data, as opposed to streaming live data
  - We can sketch a general client-server architecture

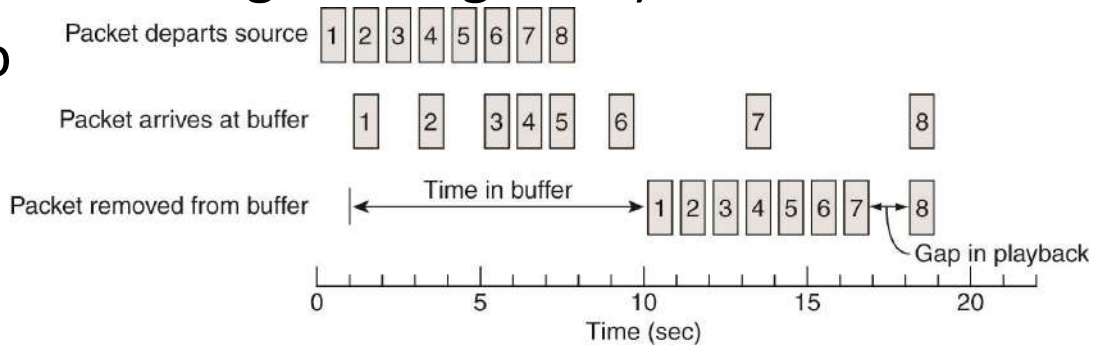


Stream-Oriented Communication *(7)*

- QoS properties from an application's perspective:
  - The required bit rate at which data should be transported
  - The max delay until a session has been set up (i.e., when an application can start sending data)
  - The max end-to-end delay (i.e., how long it will take until a data unit makes it to a recipient)
  - The max delay variance, or jitter
  - The max round-trip delay
- When dealing with stream-oriented comm. that is based on the Internet protocol stack, the basis of comm. is formed by an extremely simple, best-effort datagram service (i.e., IP)

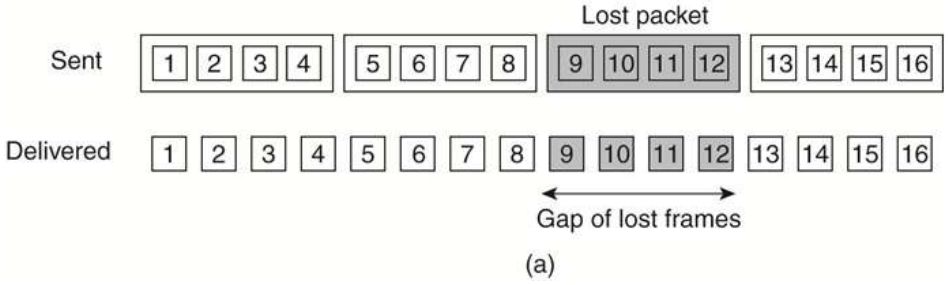
Stream-Oriented Communication *(8)*

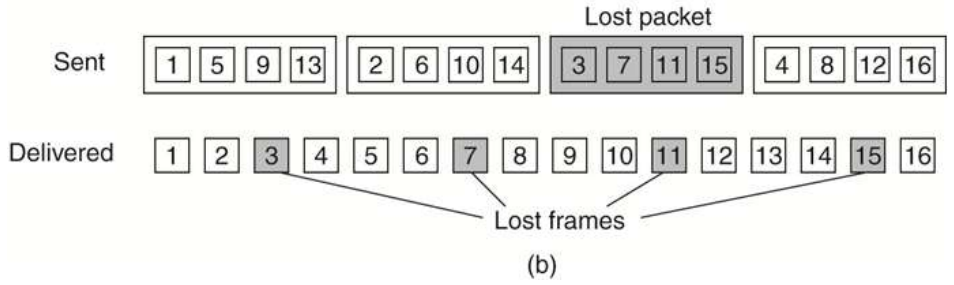
- Given that the underlying system offers only a best-effort delivery service, a DS can try to conceal as much as possible of the *lack* of QoS
  - The Internet provides a means for differentiated services; a sending host can essentially mark outgoing packets as belonging to one of several classes, including an expedited forwarding class and an assured forwarding class (by which traffic is divided into four subclasses along with three ways to drop packets if the network gets congested)
  - DS may use buffers to reduce jitter



Stream-Oriented Communication *(9)*

- Given that … (cont'd)
  - Applying forward error correction (FEC) techniques to compensate for the loss in QoS
  - Since a single packet may contain multiple audio and video frames, when it is lost, the receiver may actually perceive a large gap when playing out frames; this can be somewhat circumvented by interleaving frames, but this approach requires a large receive buffer and thus imposes a higher start delay for the receiving application



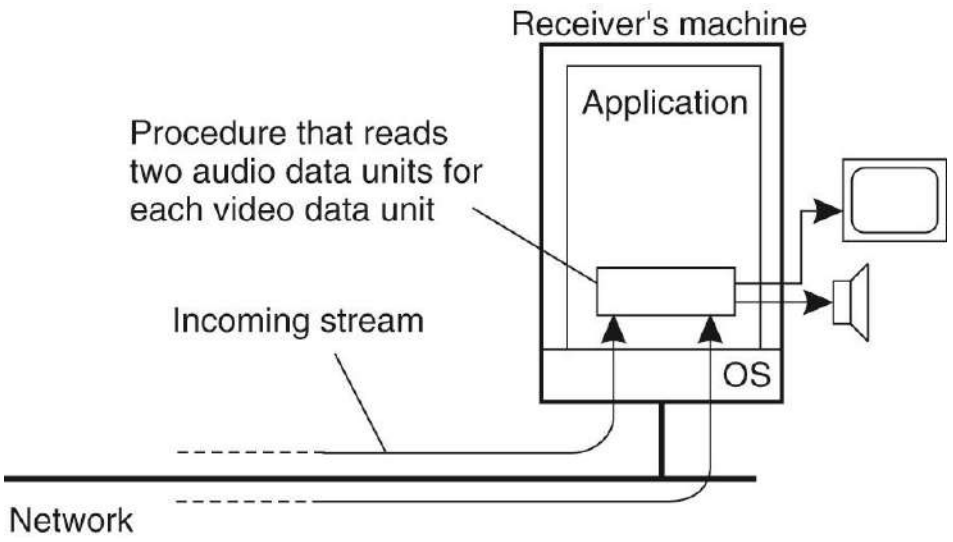


Stream-Oriented Communication *(10)*

- Stream synchronization
  - Synchronization of streams deals with maintaining temporal relations between streams
    - Sync. between a discrete data stream & a continuous data stream, e.g., a slide show enhanced with audio
    - Sync. between continuous data streams, e.g., a video stream synchronized with the audio (lip synchronization), a stereo audio stream consisting two substreams
  - Two issues in stream synchronization:
    - The basic mechanisms for synchronizing two streams
    - The distribution of those mechanisms in a networked environment
  - Sync. mechanisms can be viewed at several different levels of abstraction

Stream-Oriented Communication *(11)*

- Stream synchronization (cont'd)
  - At the lowest level, sync. is done explicitly by operating on the data units of simple streams
    - There is a process that simply executes read and write operations on several simple streams, ensuring that those operations adhere to specific timing and sync. constraints



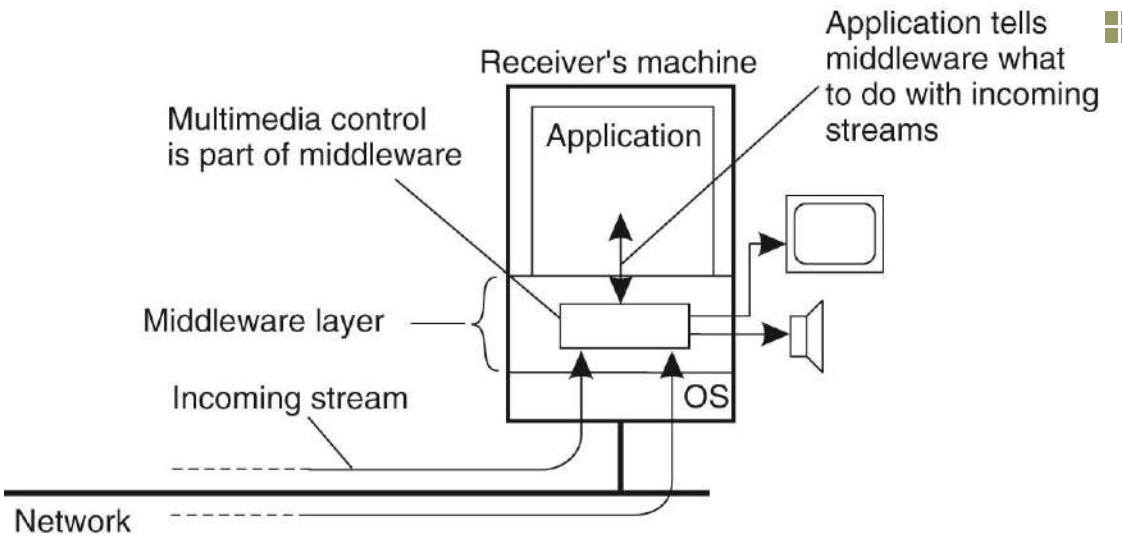
- Consider a movie presented as 2 input streams (video & audio)
  - The video stream contains uncompressed low-quality images of 320×240 pixels, each encoded by a single byte, leading to video data units of 76,800 bytes each; the images are to be displayed at 30 Hz, or one image every 33 ms

Stream-Oriented Communication *(12)*

- Stream synchronization (cont'd)
  - At the lowest level … (cont'd)
    - Consider a movie … (cont'd)
      - The 44.1 kHz, 16-bit, stereo audio stream contains samples grouped into units of 5,880 bytes, each corresponding to 33 ms of audio
      - If the input process can handle 2.5 Mbps, we can achieve lip sync. by simply alternating between reading an image and reading a block of audio samples every 33 ms; the drawback of this approach is that the application is made completely responsible for implementing sync. while it has only low-level facilities available
    - A better approach is to offer an interface that allows the application to more easily control streams and devices
    - An application developer can write a simple monitor program consisting of two handlers, one for each stream, that jointly check if the video and audio streams are sufficiently synchronized, and if needed, adjust the rate at which video or audio units are presented

Stream-Oriented Communication *(13)*

- Stream synchronization (cont'd)
  - At the lowest level … (cont'd)
    - The last example is typical for multimedia middleware systems
    - The middleware offers a collection of interfaces for controlling audio and video streams, including interfaces for controlling devices, e.g., monitors, cameras, microphones, etc.



- Each device and stream has its own high-level interfaces, including interfaces for notifying an application when some event occurred; the latter interfaces are subsequently used to write handlers for synchronizing streams

Stream-Oriented Communication *(14)*

- Stream synchronization (cont'd)
  - The distribution of synchronization mechanisms
    - The receiving side of a complex stream needs to know exactly what to do; it must have a complete *synchronization specification* locally available
      - Common practice is to provide this info implicitly by multiplexing the different streams into a single stream containing all data units, including those for sync.
      - For an example: the MPEG-2 standard allows an unlimited number of continuous and discrete streams to be merged into a single stream
    - Another issue is whether sync. should take place at the sending or receiving side
      - If the sender handles sync., it may merge streams into a single stream with a different type of data units, e.g., a stereo audio stream
      - The receiver merely has to read in a data unit, and split it into a left and right sample; delays for both channels are now identical