2. Interprocess Communication

Sistemes Distribuïts en Xarxa (SDX)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2024/2025 Q2

Contents

- **Introduction**
- Remote procedure call
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

Interprocess communication

- At the heart of every distributed system
- **<u>Low-level message passing</u>** offered by the underlying network supports communication
  - Message-passing facilities are very primitive
  - Developing large-scale systems is too difficult
  - We need communication paradigms with higher-abstraction level
- Communicating processes must adhere to a set of rules $\Rightarrow$ **protocols**

Contents

Types of communication

A. <u>Direct communication</u>
   – Senders explicitly direct messages/invocations to the associated receivers
   – Senders must know receivers' identity and both must be active at same time
   – e.g. sockets, remote invocations

B. <u>Indirect communication</u>
   – Communication through an intermediary with no direct coupling between senders and receivers
   – Indirect communication allows time and/or space **decoupling** between senders and receivers

Types of communication

A. Space decoupling
   - Sender and receiver **do not need to know the identity** of each other to communicate
   - e.g. publish-subscribe systems

B. Time decoupling
   - Sender and receiver **do not need to exist** at the same time to communicate
   - e.g. message queues
   - Time decoupling also allows distinguishing transient from persistent communication

Types of communication

A. Transient communication
   - A message is **discarded** unless the receiver is active at the time of the message delivery
   - e.g. sockets, remote invocations

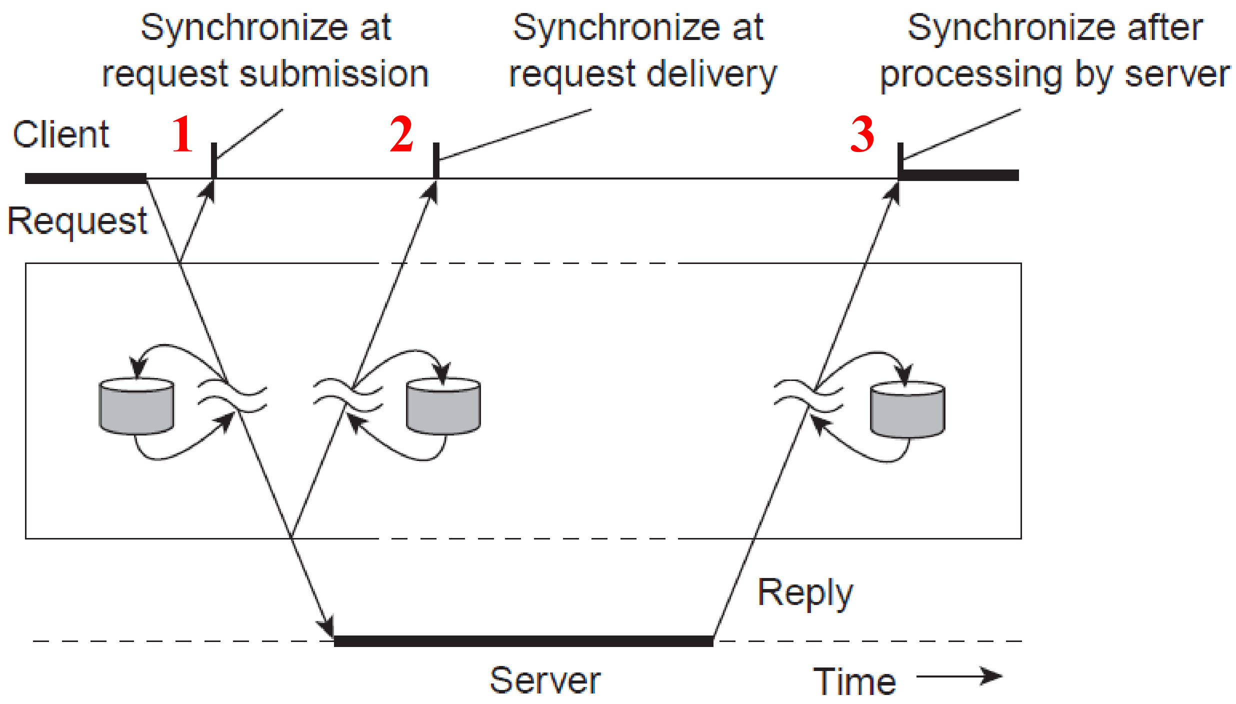
B. Persistent communication
   - A message is **stored** by the middleware as long as it takes to deliver it at the receiver
   - Sender and receiver are decoupled in time
   - e.g. message queues

Types of communication

A. Synchronous communication
   - *send* and *receive* are **blocking** operations
     - Sender is blocked until it gets a reply to its message
     - Receiver is blocked until a message arrives
   - Default for request-reply paradigms (e.g. RPC)

B. Asynchronous communication
   - *send* is a **non-blocking** operation
     - Sender proceeds immediately upon sending a message
   - *receive* can be blocking or not (i.e. get notified, by polling or interrupt, when a message arrives)
   - e.g. publish-subscribe systems, message queues

Synchronous communication



1. **Submission-based:** Block until the middleware notifies that it will take over transmission of the request
2. **Delivery-based:** Block until request is delivered to recipient
3. **Response-based:** Block until recipient replies with response

Types of communication

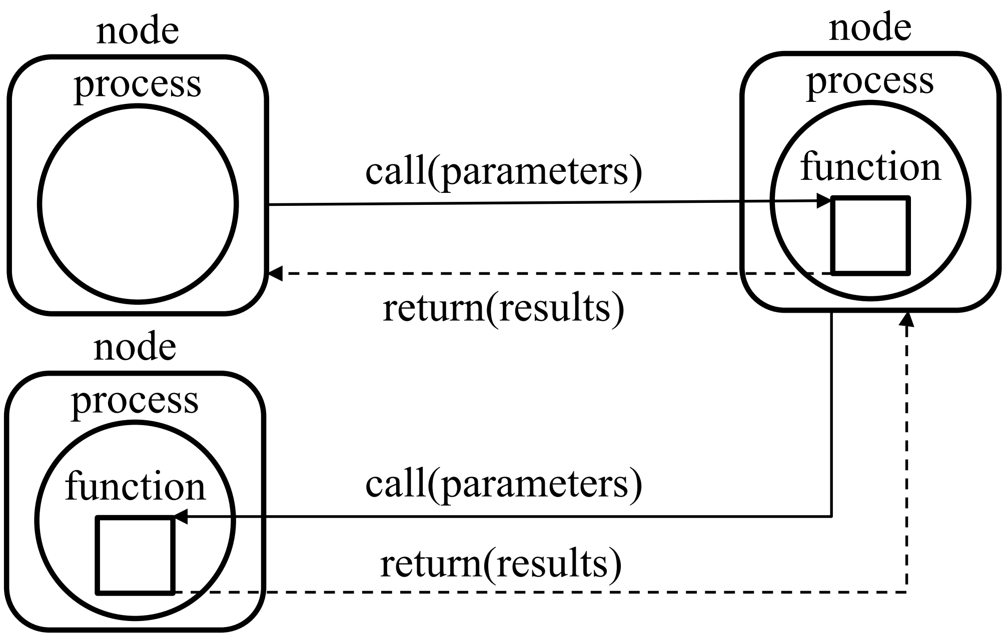
A. Discrete communication
   - Exchange of 'independent' units of information
   - Timing has no effect on correctness
   - e.g. sockets, message queues, remote invocations

B. Continuous communication
   - Messages are related to each other by the order they are sent, or by a temporal relationship
   - Timing between data items must be preserved to interpret correctly the data
   - e.g. streams (audio, video, …)

Contents

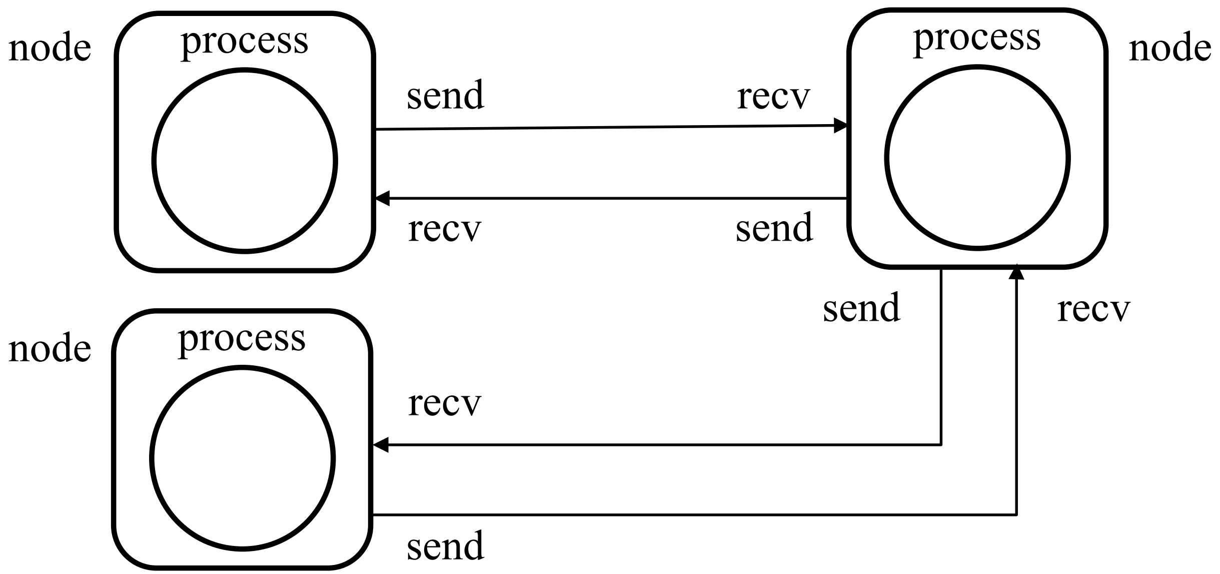
Remote invocation

- Transparent extension to traditional programming: a node can call a function in another one as if it was local (e.g. RPC)
- Direct, transient, synchronous point-to-point interactions
- Middleware handles the marshaling/unmarshaling of parameters and can implement delivery guarantees



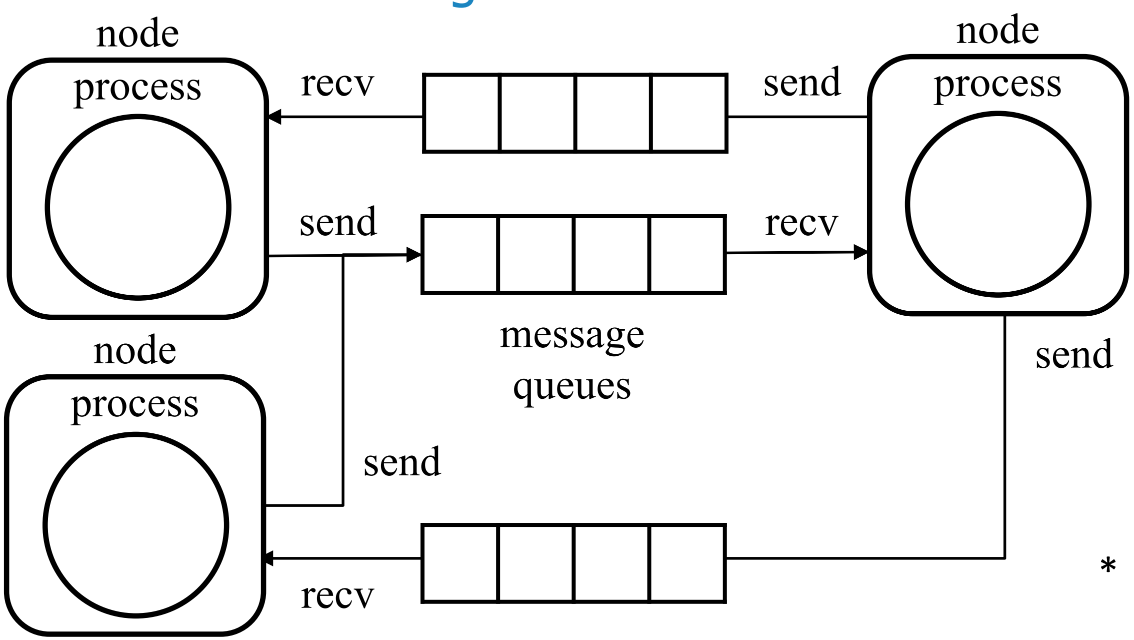
Message passing

- Direct, transient networked communication between processes
  - e.g. sockets, Message Passing Interface (MPI), ZeroMQ
- Generally synchronous and point-to-point
  - MPI and ZeroMQ also support asynchronous and multipoint communication
- Not mediated by the middleware
- Lacks transparency: exposes the network characteristics/issues



Message queuing

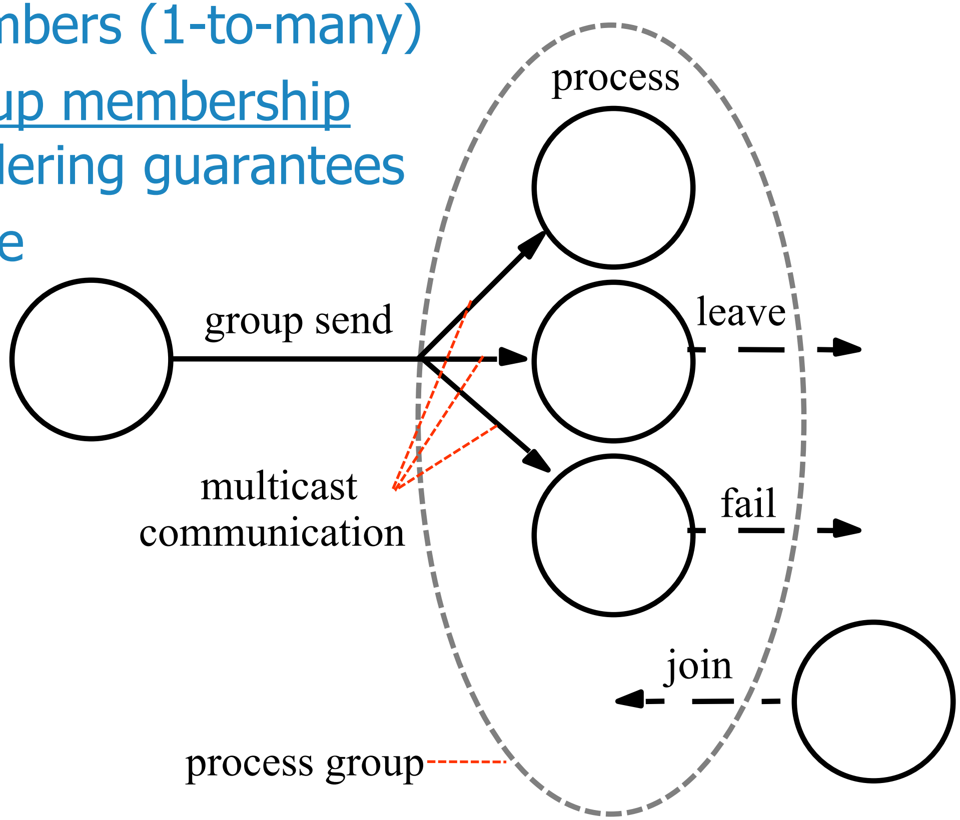
- Sender puts message into a queue, receiver gets it from queue
  - e.g. e-mail, Message Oriented Middleware (Apache ActiveMQ*, RabbitMQ*)
- Asynchronous and persistent (space decoupling is possible)
- Point-to-point (sender ⇔ queue ⇔ receiver)
- Middleware stores messages and forwards them across queues



* support also for publish-subscribe

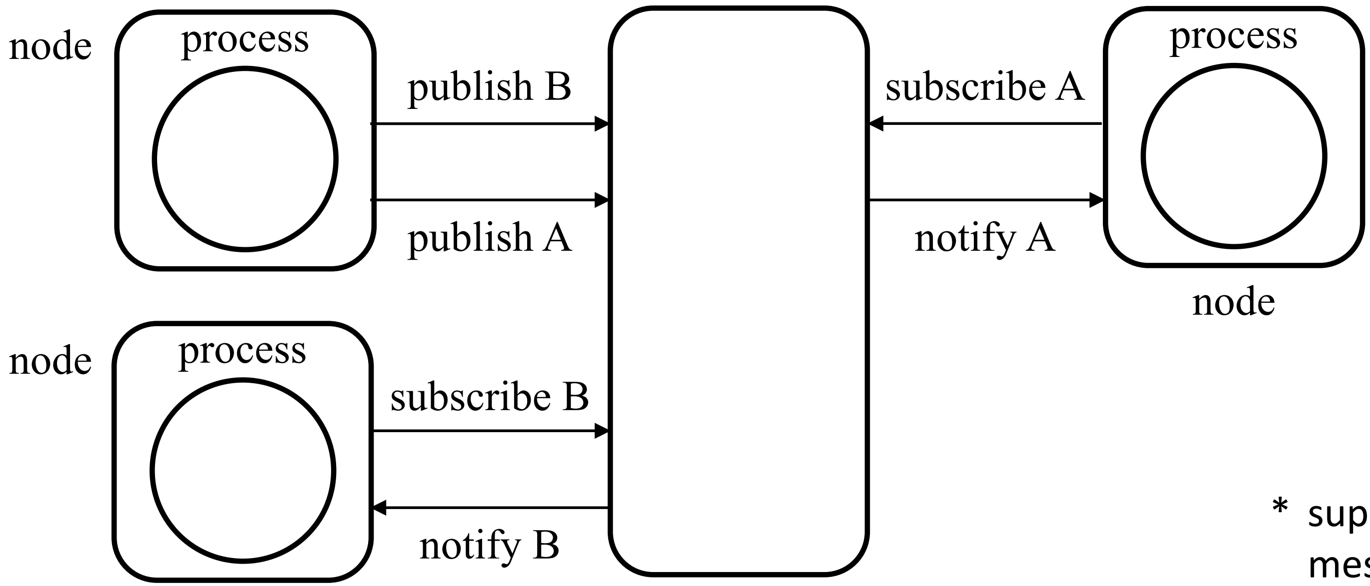
Group communication

- Messages sent to a group via the group identifier: do not need to know the recipients (communication is space decoupled)
- Delivered to all group members (1-to-many)
- Middleware maintains group membership and provides reliability/ordering guarantees
- Time decoupling is possible
- e.g. JGroups, Spread



Publish-subscribe

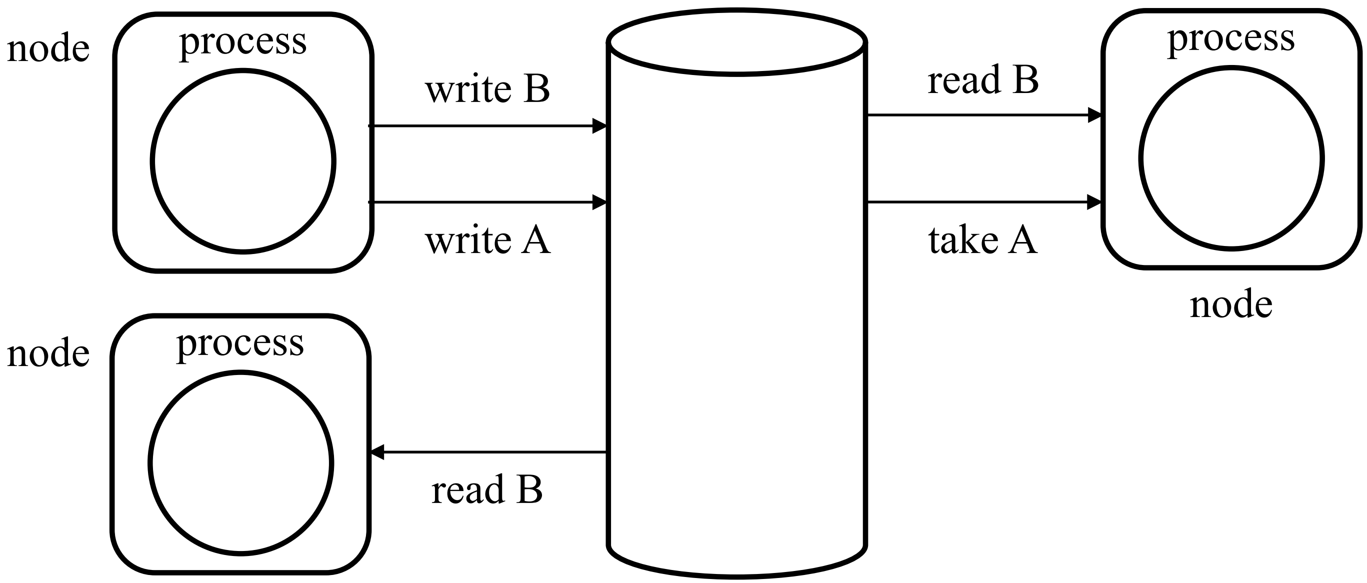
- Asynchronous 1-to-many communication by propagating events
  - Producers publish structured events, consumers express interest in events through subscriptions (e.g. Apache Kafka*, Apache ActiveMQ*, Scribe)
- Space decoupled (time decoupling is possible)
- Middleware efficiently matches subscriptions against published events and ensures the correct delivery of event notifications



* support also for message queues

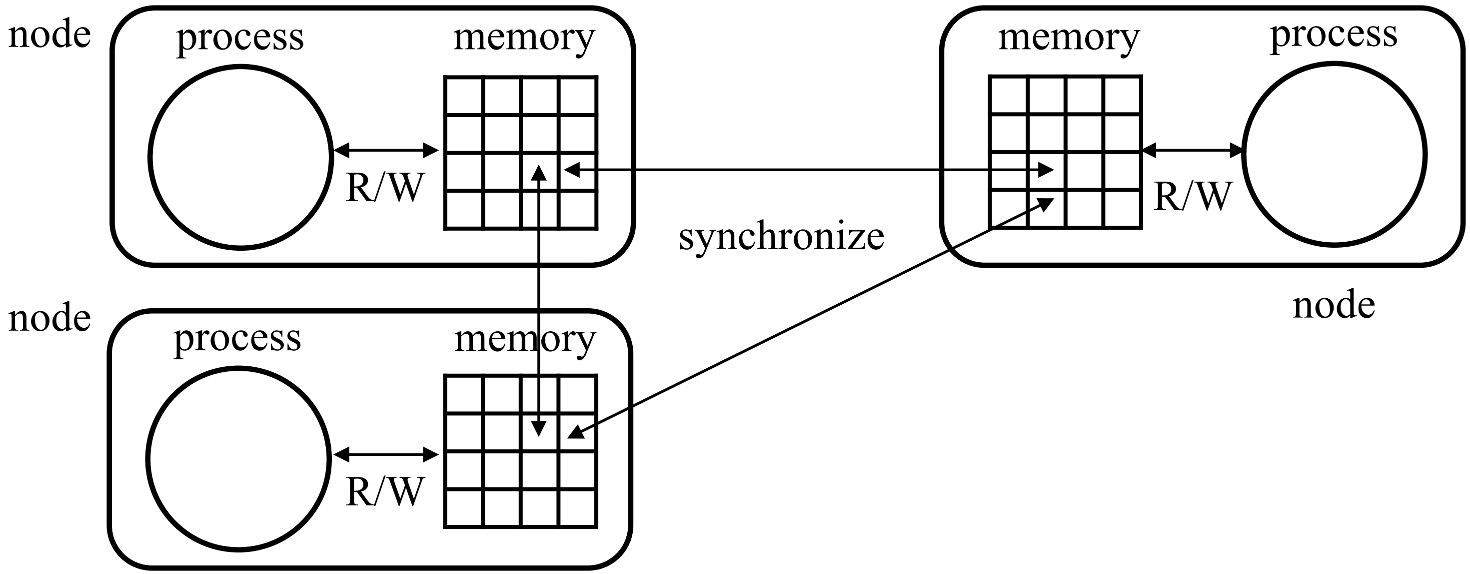
Shared data space

- Persistent, asynchronous communication using a shared storage
  - e.g. JavaSpaces, which also provide space decoupling by means of pattern matching on contents
- Post items to shared space; consumers can read (1-to-many) or take (point-to-point) them at a later time
- Middleware keeps the data space



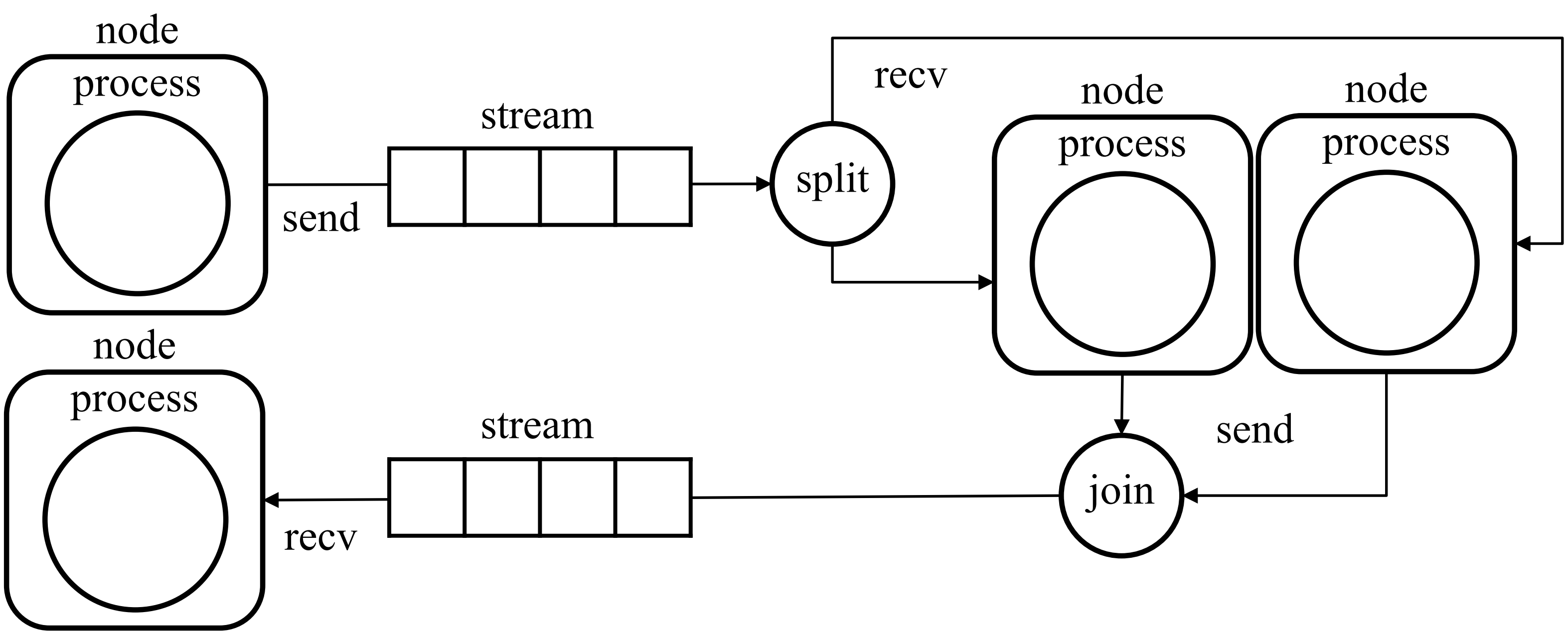
Shared memory

- Share data between processes as if they were in their own local address spaces (extend traditional parallel programming model)
  - e.g. distributed shared memory (DSM): Treadmarks, Linda, Orca
- Space decoupled (time decoupling is possible)
- Interaction is multipoint (many nodes share memory)
- Middleware synchronizes and maintains the consistency of data



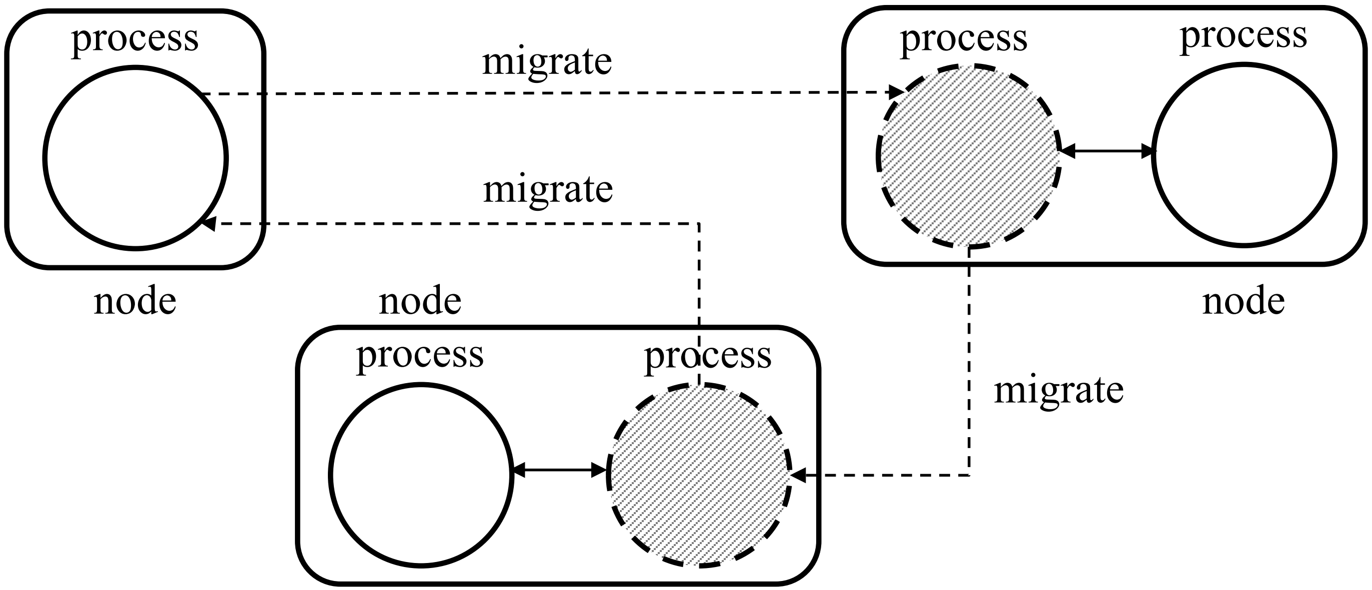
Stream oriented

- Processing of large sequences of continuous data streams
  - e.g. distributed multimedia applications (video, audio), sensor data
- Direct, transient, 1-to-many communication
- Middleware ensures timing, coordinates flows (splits, joins) and handles issues such as congestion, delays, and failures



Mobile code / agents

- Code or running processes travel from one node to another and interact locally with other components
  - e.g. code: web applets, JavaScript, Flash, ActiveX
  - e.g. agent: Java Agent Development Framework (JADE), VM migration
- Local interactions are faster, but incur potential security threats
- Middleware transfers code and saves/restores process state



Contents

Remote Procedure Call (RPC)

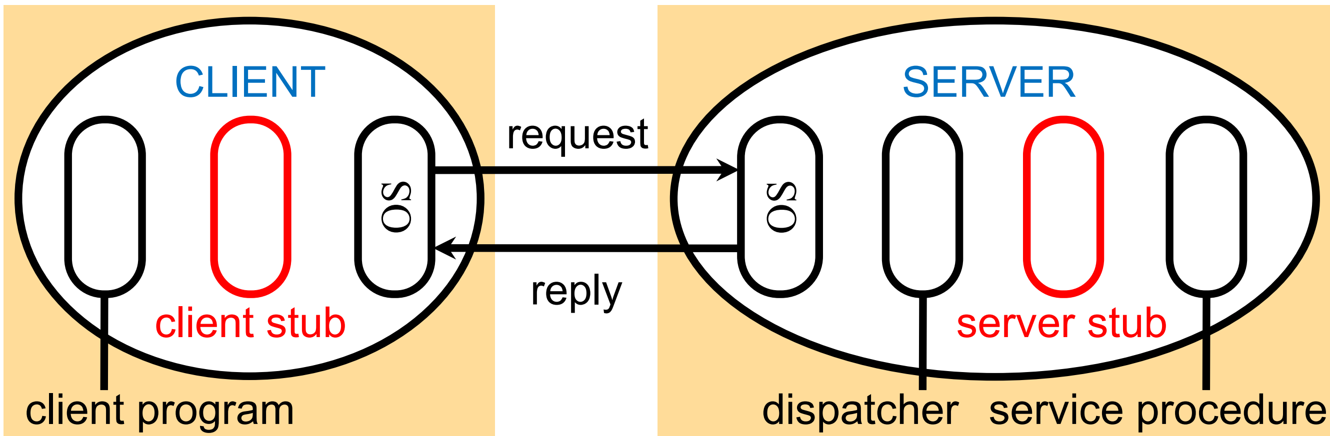
- Procedures located on remote machines can be called as if they were local
  - RPCs are **transparent** as they hide the details of network programming (i.e. no more sockets)
- Conceptually simple, but …
  - Machines may have different architectures and caller & callee have different address spaces
  - How are parameters/results (of different types) passed to/from a remote procedure?
  - What happens if one or both of the machines crash while the procedure is being called?

Contents

- Introduction
- **Remote procedure call**
  - **Basic RPC operation**
  - RPC parameter passing
  - Extended RPC models
  - RPC semantics and failures
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

Basic RPC operation

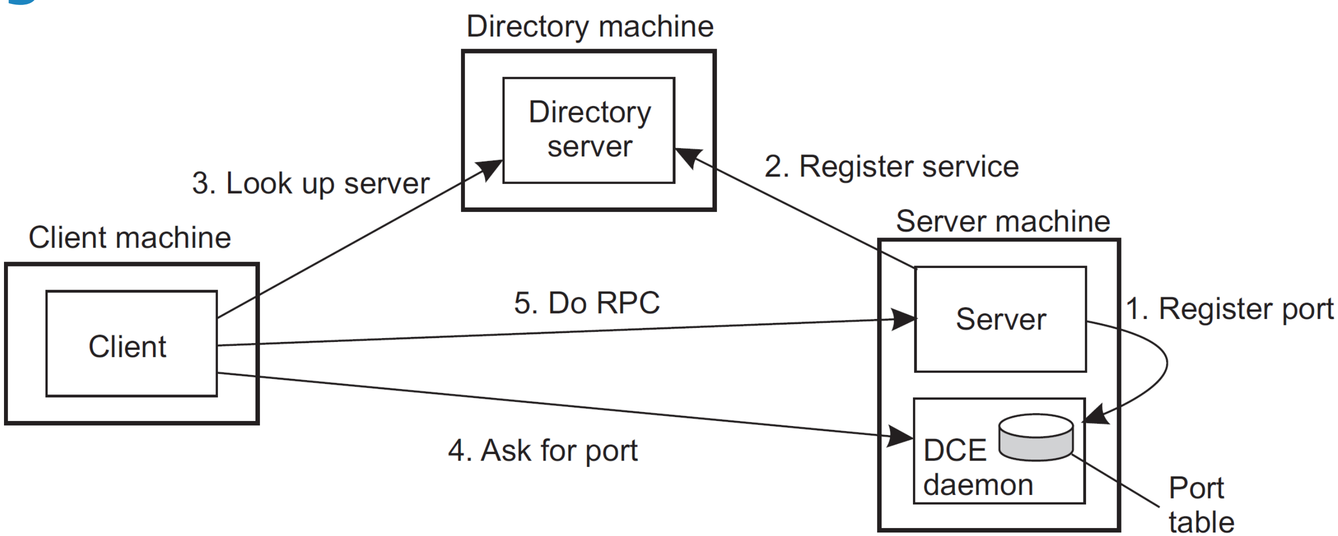
- Transparency is achieved by using **stubs**



1. Client program calls client stub normally
2. Client stub packs arguments into a request message and calls local OS
3. Client OS sends message to server
4. Server OS receives message, which is dispatched to corresponding server stub
5. Server stub unpacks parameters and calls service procedure
6. Procedure runs and returns result to server stub
7. Server stub packs result in a reply message and calls local OS
8. Server OS sends message to client
9. Client OS gives message to client stub
10. Client stub unpacks result and returns to client program

Basic RPC operation

- Client-to-server binding
  1. Find server's machine ⇒ use a <u>directory service</u>
  2. Discover the server's port on that machine ⇒ ask the RPC daemon (running at a well-known port)
- e.g. DCE RPC

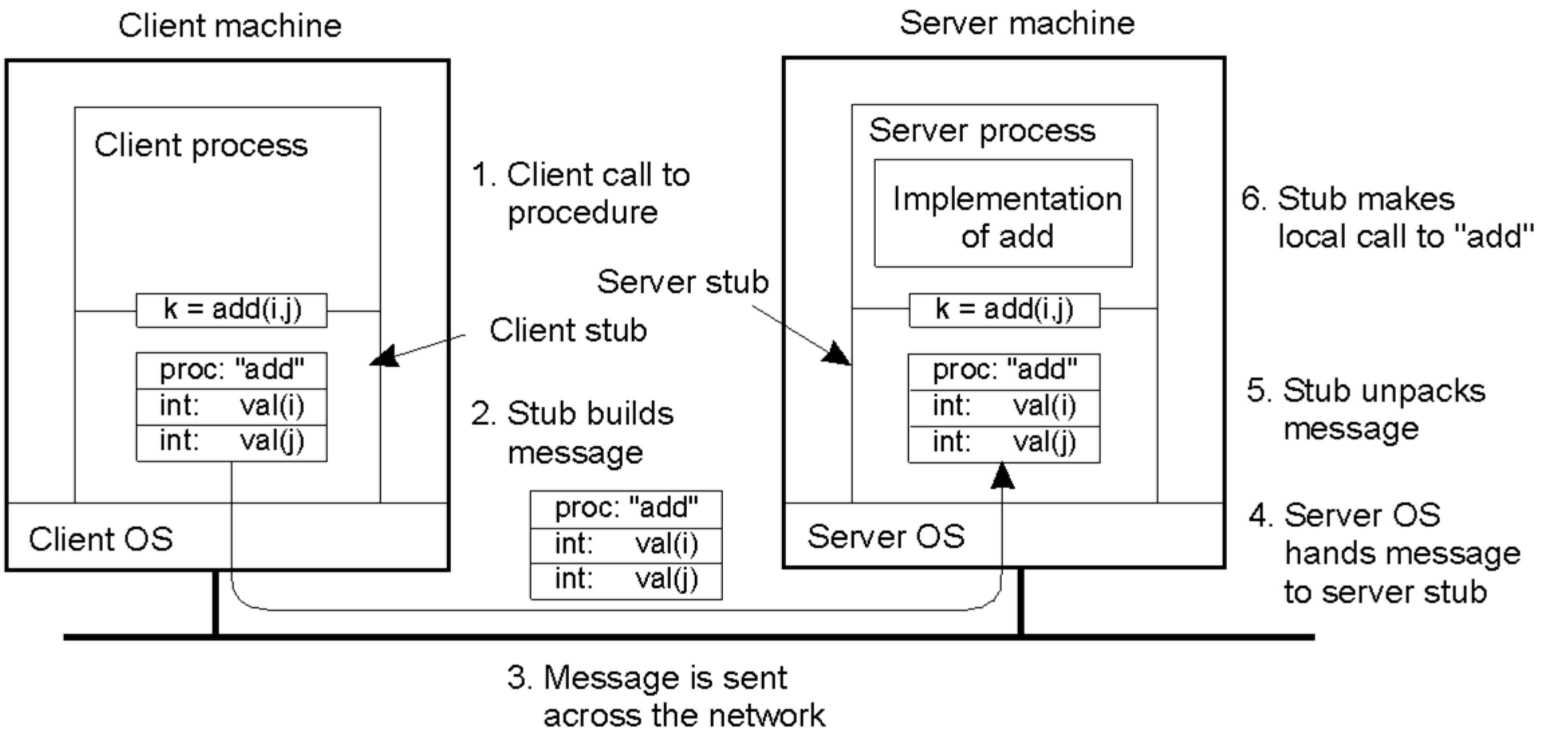


Contents

- Introduction
- **Remote procedure call**
  - Basic RPC operation
  - **RPC parameter passing**
  - Extended RPC models
  - RPC semantics and failures
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

RPC parameter passing

- Stubs take care of parameter **marshalling**
  - Transform parameters/results into a byte stream, which is sent across the network



RPC parameter passing

- Passing value parameters
  - Works well if the machines are homogeneous
  - Complications arise when the two machines …
    - use different character encodings: ASCII vs. EBCDIC
    - use different byte-ordering: little-endian vs. big-endian
  - Solution: Both communicating parties agree on the same message data format
    - Transform data to be sent to a standard machine- and network-independent format
    - Marshaling/unmarshaling is all about this transformation
    - e.g. JSON: JavaScript Object Notation, XML: Extensible Markup Language, XDR: External Data Representation

RPC parameter passing

- Passing reference parameters
  - Pointers are meaningful only within a process address space
  - By default, RPC does not offer call by reference
  - Some RPC can pass by reference <u>arrays (of known length) & structures</u> $\Rightarrow$ **copy/restore semantics**
    - Client stub copies the entire data structure to network
    - Server stub saves it to memory and passes a pointer
    - IN/OUT markers may eliminate one copy operation
  - Some languages can support <u>global references</u>

Interface Definition Language (IDL)

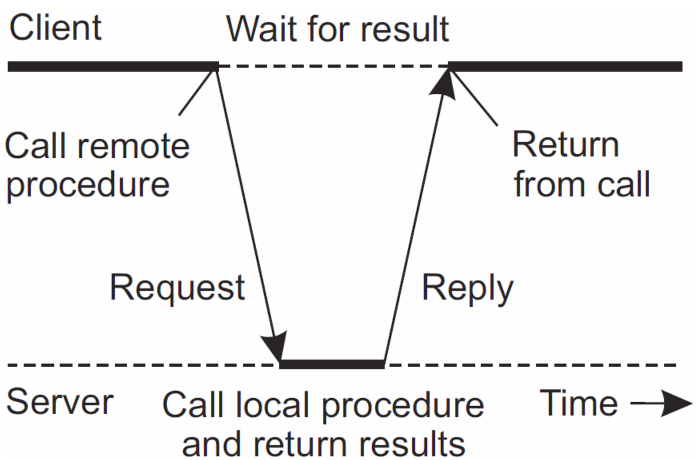
- Definition of interfaces simplifies RPCs
  - Service interface specifies the procedures offered by a server, defining the types of the parameters of each of the procedures
- Interfaces are defined by means of an IDL
  - e.g. WSDL (Web Services); Protobuf (Google)
  - IDLs are language-neutral
    - Don't presuppose the use of any programming language
  - Stub compiler generates stubs automatically from specs in an IDL
    - e.g. rpcgen tool (SUN RPC)

Contents

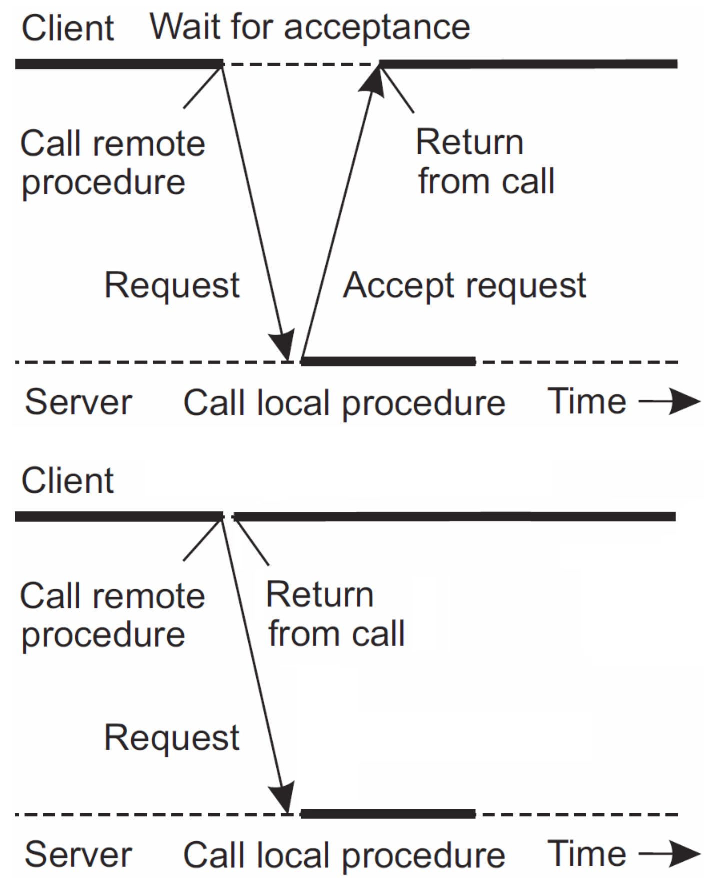
- Introduction
- **Remote procedure call**
  - Basic RPC operation
  - RPC parameter passing
  - **Extended RPC models**
  - RPC semantics and failures
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

Extended RPC models: Asynchronous RPC

Normal RPC



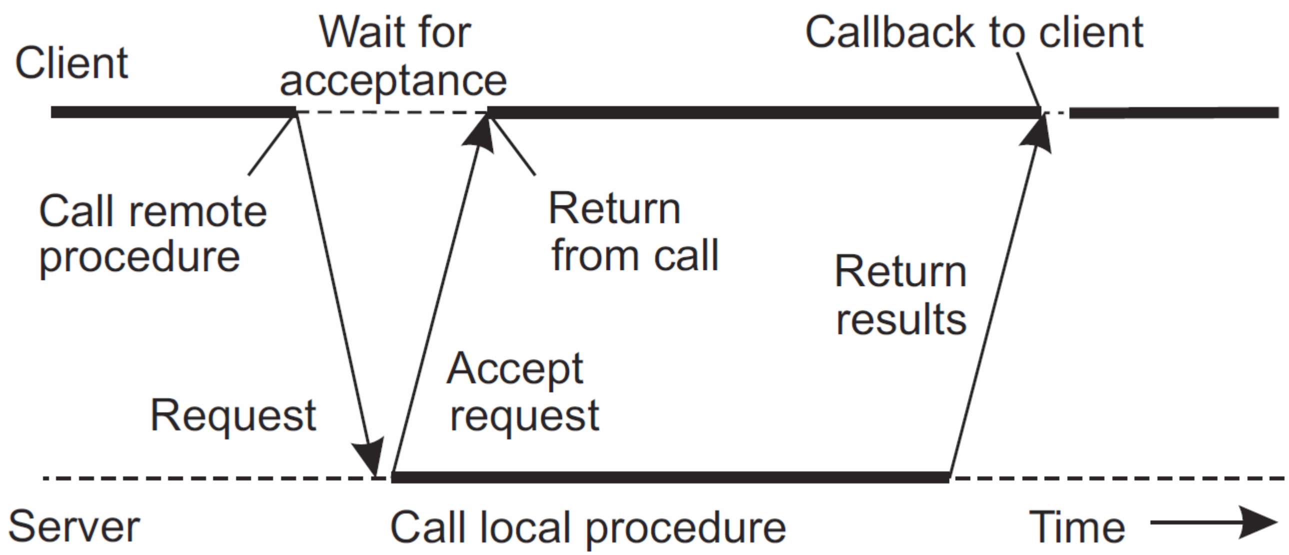
Asynchronous RPC / One-way RPC



- Useful when the client does not need or expect a result

Extended RPC models: Deferred Synchronous RPC

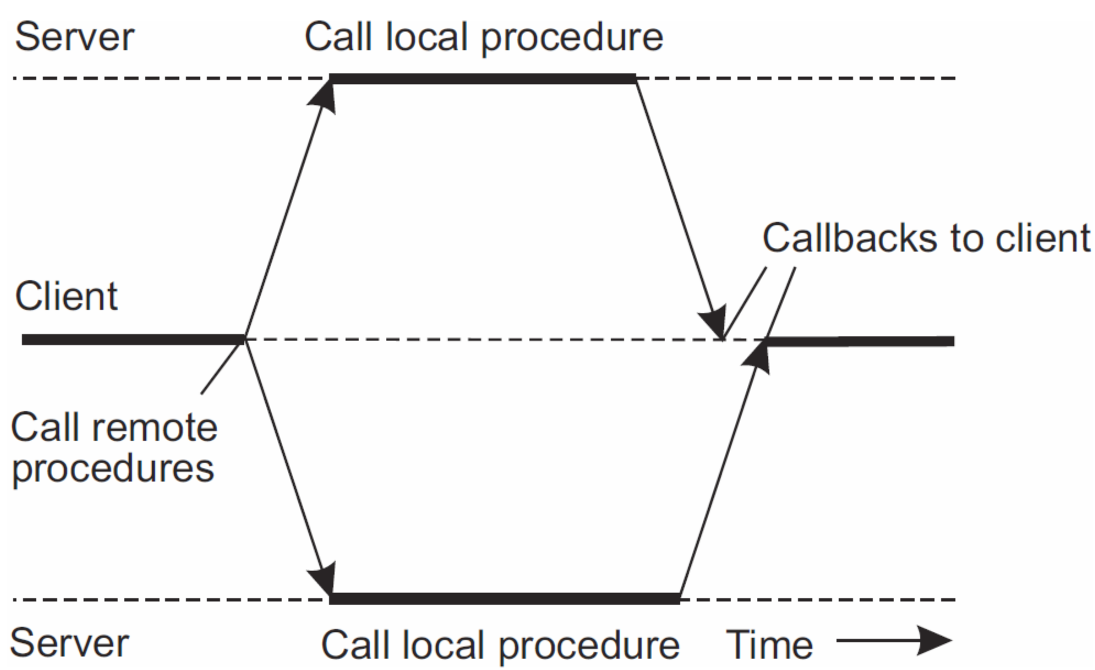
- Communication through an asynchronous RPC and a **callback**



↑ Allows a client to perform other useful work while waiting for the results

Extended RPC models: Multicast RPC

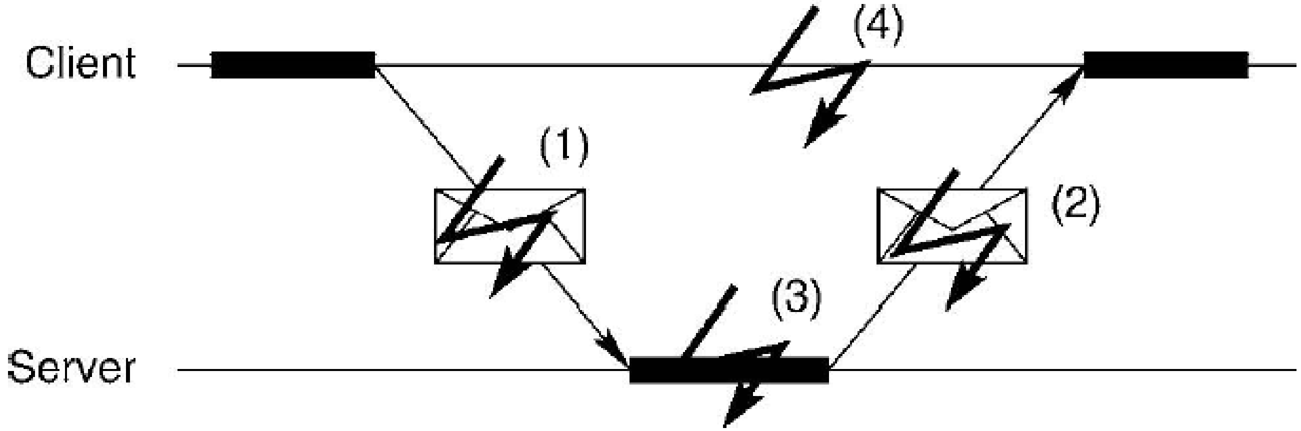
- Execute multiple RPCs at the same time using one-way RPCs and callbacks



Contents

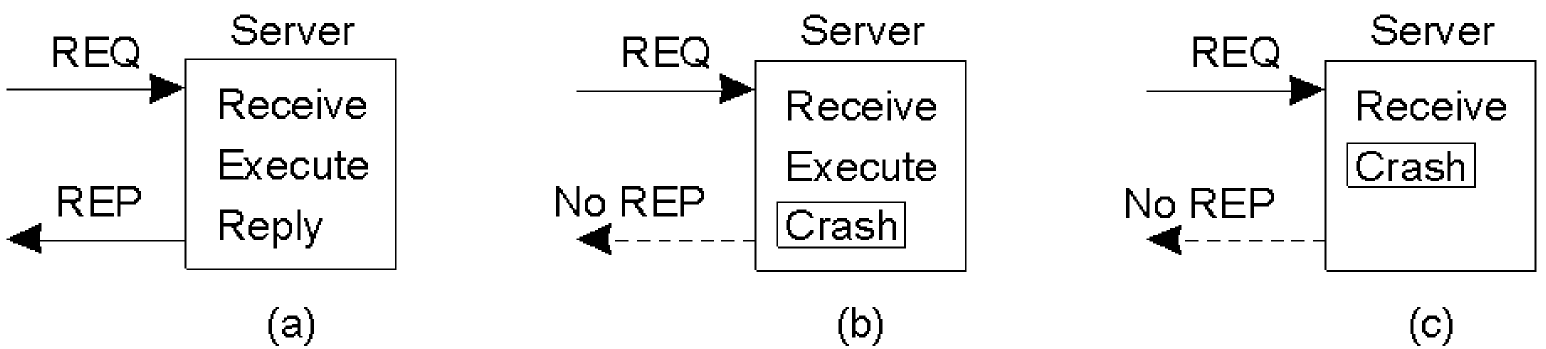
RPC semantics and failures

- RPCs work well as long as client and server function perfectly. 4 types of RPC failures:
  1) Client's request is lost
  2) Server's reply is lost
  3) Server crashes after receiving a request
  4) Client crashes after sending a request



Handling (1), (2), and (3)

- PROBLEM: Client perceives (1), (2), and (3) identically ⇒ Reply message does not arrive
- WARNING: b) and c) require different handling but client cannot distinguish them



- Server must also participate in the techniques to handle correctly these situations

Handling (1), (2), (3): Techniques

A. Retry request message
   - Client sets a timeout when it is waiting to get the server's reply message
   - After a timeout, client retransmits the request until either a reply is received or the server is assumed to have failed

B. Duplicate filtering
   - Client assigns a unique identifier to each request
   - Server filters out duplicate requests to avoid re-executing the operations
     - This requires the server to be stateful

Handling (1), (2), (3): Techniques

C. Retransmission of results
   - Server keeps a history of prior results to resend lost replies without re-executing the operations
     - This requires the server to be stateful
   - How to avoid the history to become huge?
     - If clients can make only one request at a time, server can interpret each request as an ACK of its prior reply
       - $\Rightarrow$ History can contain only the last reply message
     - Messages are also discarded after a period of time

D. Recoverable processes
   - A process is automatically restarted after a crash and can recover its state from persistent storage

Handling (1), (2), (3): Semantics

- Combinations of these techniques lead to several semantics for the reliability of RPCs:

a) Maybe semantics

  - RPC may be executed <u>once or not at all</u> in case of lost request or reply messages or server crash
  - Request message is sent only once $\Rightarrow$ does not use any technique to tolerate failures
  - Useful only for applications in which occasional failed RPCs are acceptable

Handling (1), (2), (3): Semantics

b) At-least-once semantics

- RPC will be executed at least once, but possibly more, in case of lost request or reply messages
- RPC may be executed several times, or possibly not at all, in case of lost messages or server crash
- Uses technique 'Retry request message'
- Can be used safely if operation is **idempotent**
  - Can be performed repeatedly with the same effect as if it had been performed exactly once
    - Pure read operations: e.g. loading a static web page
    - Strict overwrite operations: e.g. update your billing address in an online shop

Handling (1), (2), (3): Semantics

c) At-most-once semantics

- RPC will be executed exactly once in case of lost request or reply messages
- RPC will be executed at most once, or possibly not at all, in case of lost messages or server crash
- Uses techniques ‘Retry request message’, ‘Duplicate filtering’, and ‘Retransmission of results’
- Appropriate for non-idempotent operations
  - e.g. electronic transfer of money

Handling (1), (2), (3): Semantics

d) Exactly-once semantics

- RPC will be executed exactly once in case of lost request or reply messages and server crash
- Uses techniques ‘Retry request message’, ‘Duplicate filtering’, ‘Retransmission of results’, and ‘Recoverable processes’
- Ideal semantics for all the applications, but difficult to achieve

Handling (4)

- A crash of the client generates '**orphan**' calls
  - RPC is active in the server but has some ancestor executing on a crashed node
- Orphan calls should be eliminated
  - They waste resources (e.g. CPU cycles)
  - They can lock resources (e.g. files, semaphores)
  - Client can confuse old replies after recovering
- We present 4 strategies, but none is perfect
  - Killing orphans can have consequences: locks held forever, traces of orphans (e.g. jobs in queues) …

Handling (4)

a) Extermination

- Client logs its RPC calls to persistent storage
- Upon recovery of a crash, client requests remote nodes to kill its orphans
- Each remote node exterminates the orphans and requests to kill their corresponding descendants

↑ Only nodes with orphans are checked and only orphan RPCs are aborted

↓ Orphans may survive when nodes fail or network is partitioned

↓ Overhead of logging (for every RPC)

Handling (4)

b) Expiration

– Each RPC is given a time limit T to complete
– Remote nodes abort RPCs when their limits expire
– If the client waits a time T after rebooting, it will guarantee that all its orphans are gone
↑ Works with network partitions and down nodes, as communication is not required to kill orphans
↓ All nodes must periodically check their RPCs
↓ Time limit must be carried in every RPC message
↓ If T is too short, non-orphans could be aborted
  - Check with the RPC owner if the deadline can be delayed

Handling (4)

c) Reincarnation

– Divide time into <u>epochs</u> (sequentially numbered)

– Upon crash recovery, client declares new epoch

  ↓ Epoch must be carried in every RPC call message (can be also broadcasted upon recovery)

– Upon receipt of new epoch, remote nodes also reincarnate and kill all RPCs from previous epoch

↑ Orphans may survive if network is partitioned, but their responses will contain an obsolete epoch number $\Rightarrow$ easily detected

↓ All nodes kill RPCs from previous epoch and non-orphan RPCs could be aborted

Handling (4)

d) Gentle reincarnation

– Like reincarnation, but upon receipt of new epoch, remote nodes also reincarnate and check with the owners of all of their RPCs

– Only when an owner does not respond, the corresponding orphans are killed

↑ Only orphan RPCs are aborted

↓ All nodes check RPCs from previous epoch

Contents

Contents

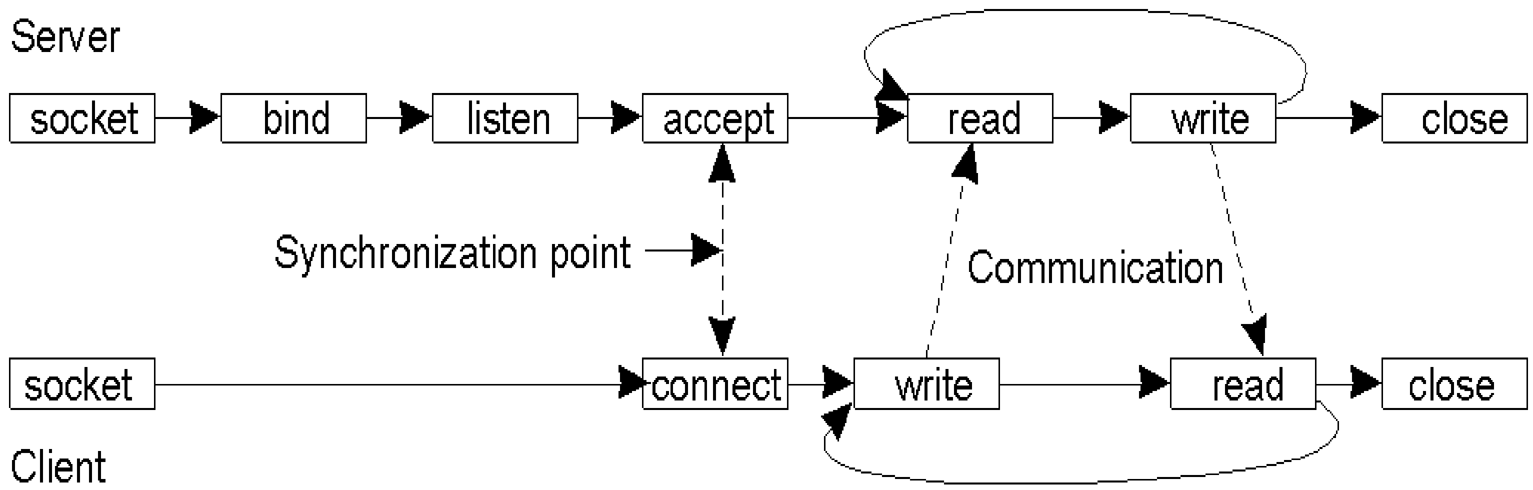
- Introduction
- Remote procedure call
- **Message-oriented communication**
  - **Message passing: sockets**
  - Message queuing: MOM
- Event-based communication
- Stream-oriented communication

Berkeley sockets

- API

Primitive	Meaning
socket	Create a new communication endpoint
bind	Attach a local address to a socket
listen	Configure the socket to accept connections and its maximum number of pending requests
accept	Block caller until a connection request arrives
connect	Actively attempt to establish a connection
write/send	Send some data over the connection
read/recv	Receive some data over the connection
close	Release the connection

Berkeley sockets



- Low level of abstraction
  - Supports only simple 'send' and 'receive' primitives
- Too closely coupled to TCP/IP networks

Berkeley sockets: example

```
int main( int argc, char *argv[] ) {
   int sfd, nsfd;
   char* msg = "Hello World!\n";
   struct sockaddr_in saddr, caddr;

   sfd = socket(AF_INET, SOCK_STREAM, 0);

   bzero((char *)&saddr, sizeof(saddr));
   saddr.sin_family = AF_INET;
   saddr.sin_addr.s_addr = htonl(INADDR_ANY);
   saddr.sin_port = htons(8000);

   bind(sfd, (struct sockaddr *)&saddr, sizeof(saddr));
   listen(sfd, 5);
   nsfd = accept(sfd, (struct sockaddr *)&caddr, &sizeof(caddr));

   send(nsfd, msg, strlen(msg), 0);
   close(nsfd);
   close(sfd);
}
```

SERVER
PROGRAM

Berkeley sockets: example

```
int main(int argc, char *argv[]) {
   int sfd, res;
   struct sockaddr_in saddr;
   char buffer[256];

   sfd = socket(AF_INET, SOCK_STREAM, 0);

   bzero((char *)&saddr, sizeof(saddr));
   saddr.sin_family = AF_INET;
   saddr.sin_addr.s_addr = inet_addr("127.0.0.1");
   saddr.sin_port = htons(8000);

   connect(sfd, (struct sockaddr*)&saddr, sizeof(saddr));

   res = recv(sfd, buffer, sizeof(buffer), 0);
   write(1, buffer, res);
   close(sfd);
}
```

CLIENT PROGRAM

Contents

Message-queuing systems

- Support persistent, asynchronous, point-to-point communication
  - Persistency requires intermediate storage for messages while sender/receiver are inactive
    - Has explicit queues that are third-party entities, separate from the sender and the receiver
  - Point-to-point in that sender places the message into a queue, and it is then removed by a single process
- a.k.a. Message-Oriented Middleware (MOM)
- e.g. e-mail, Apache ActiveMQ, RabbitMQ

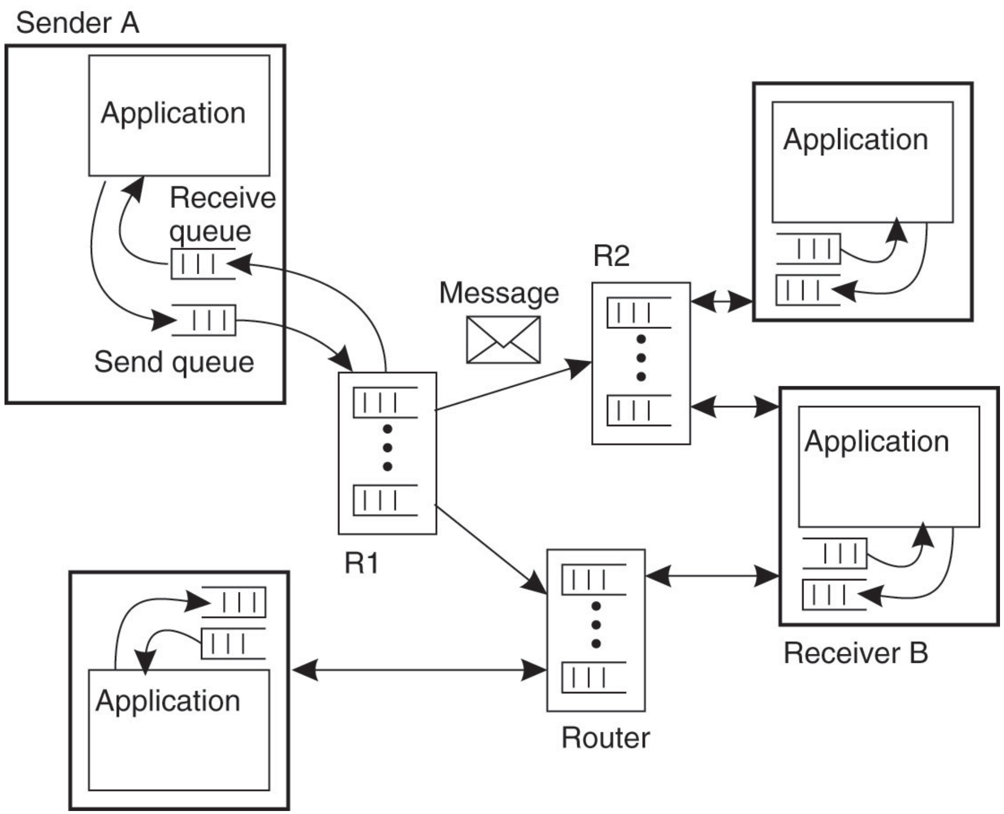
Message-queuing systems

- API
  - Very simple, yet extremely powerful abstraction

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

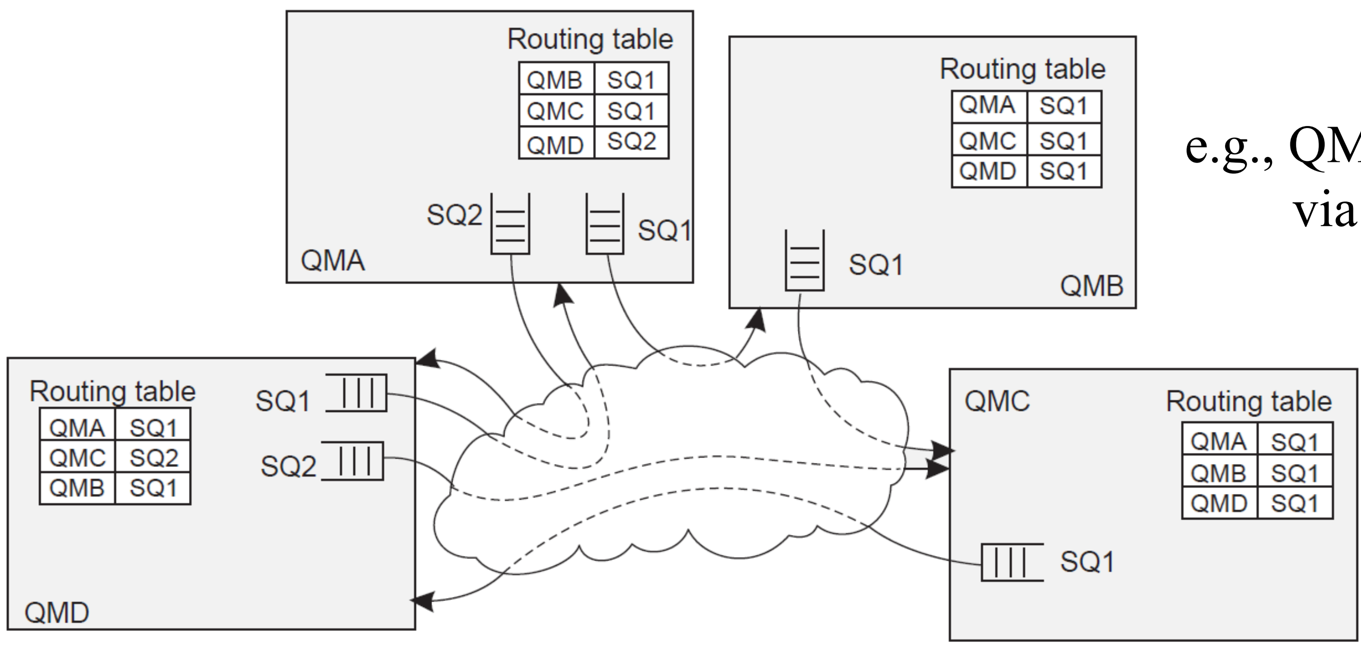
Message-queuing systems

- Applications communicate by putting/taking messages into/out from 'message queues'
- Queue Managers route messages from source to destination queue
  - Can interact directly with sender/receiver or operate as *relay* forwarding messages to other queue managers (QM)



Message-queuing systems: routing

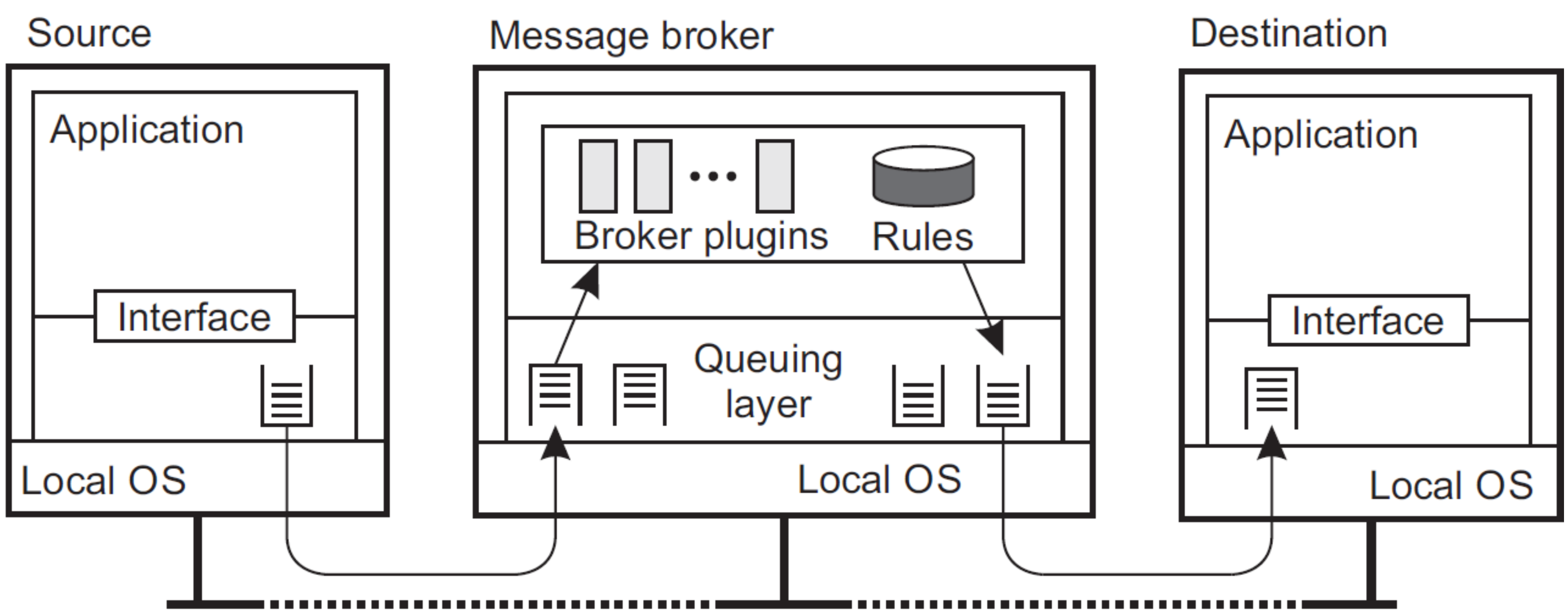
- QMs have logical, location-independent names
- Each QM maps names-to-addresses in a routing table
- As QM form an overlay network, each QM only knows to which adjacent QM it must forward a message



e.g., QMA to QMC via QMB

Message-queuing systems

- Applications must understand messages they receive
  a) Agree on a common message format (i.e. structure and data representation)
  b) Add message brokers that convert incoming messages to target format

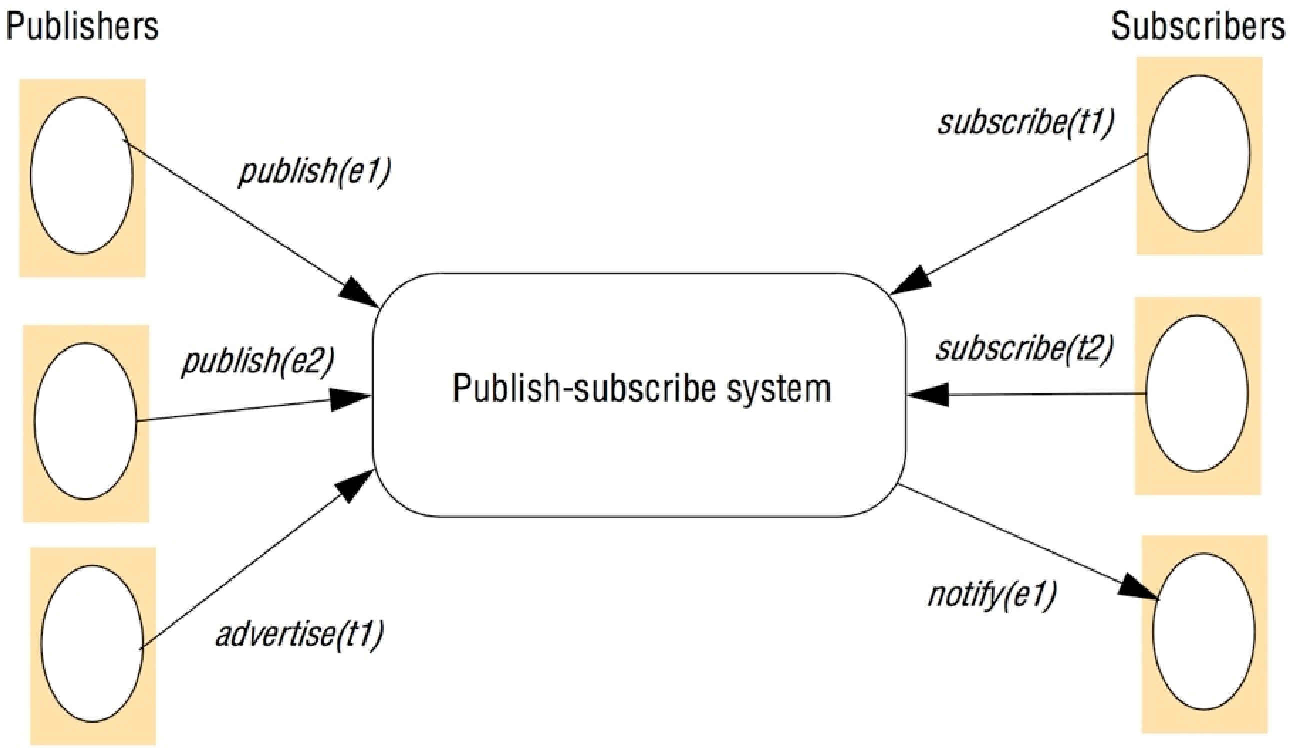


Contents

- Introduction
- Remote procedure call
- Message-oriented communication
- **Event-based communication**
- Stream-oriented communication

Publish-subscribe systems

- Publishers publish **structured events**
- Subscribers **express interest** in particular events
- Middleware matches published events to subscriptions



Publish-subscribe systems

- API

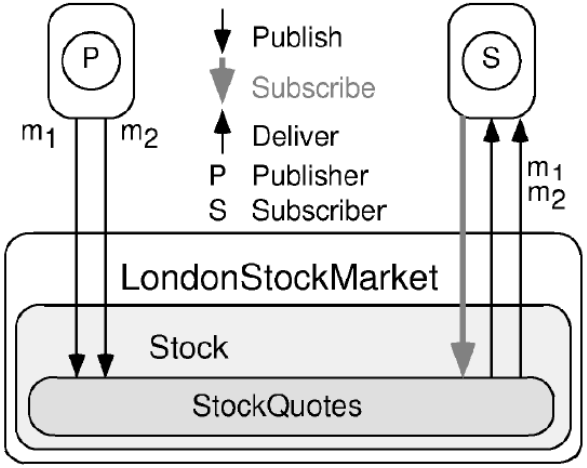
Primitive	Meaning
publish	Disseminate an event
subscribe	Express an interest in a set of events via a filter (pattern over all possible events)
notify	Deliver event
unsubscribe	Revoke interest in events
advertise	Advertise the nature of future events
unadvertise	Revoke an advertisement

Subscription models

1. **Topic-based** (a.k.a. **subject-based**)

- Subscriptions defined in terms of the topic of interest (identified by keywords)
  - '/LondonStockMarket/Stock/StockQuotes'
- Topics can be hierarchically organized

↓ Limited expressiveness

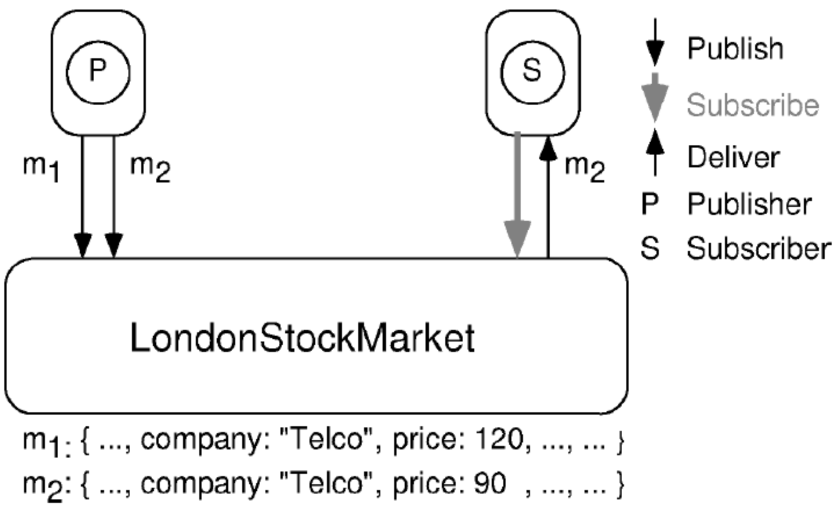



```
public class StockQuoteSubscriber implements Subscriber {
  public void notify(Object o) {
    if (((StockQuote)o).company == 'TELCO' && ((StockQuote)o).price < 100)
      buy();
  }
}
// ...
Topic quotes = EventService.connect("/LondonStockMarket/Stock/StockQuotes");
Subscriber sub = new StockQuoteSubscriber();
quotes.subscribe(sub);
```

Subscription models

2. Content-based

- Subscribe via compositions of constraints over the values of event attributes
  - Stock quote: (company == 'TELCO') and (price < 100)
- More expressive (↑), costly event matching (↓)




```
public class StockQuoteSubscriber implements Subscriber{
  public void notify(Object o) {
    buy();      // company == 'TELCO' and price < 100
  }
}
// ...
String criteria = ("company == 'TELCO' and price < 100");
Subscriber sub = new StockQuoteSubscriber();
EventService.subscribe(sub, criteria);
```

Subscription models

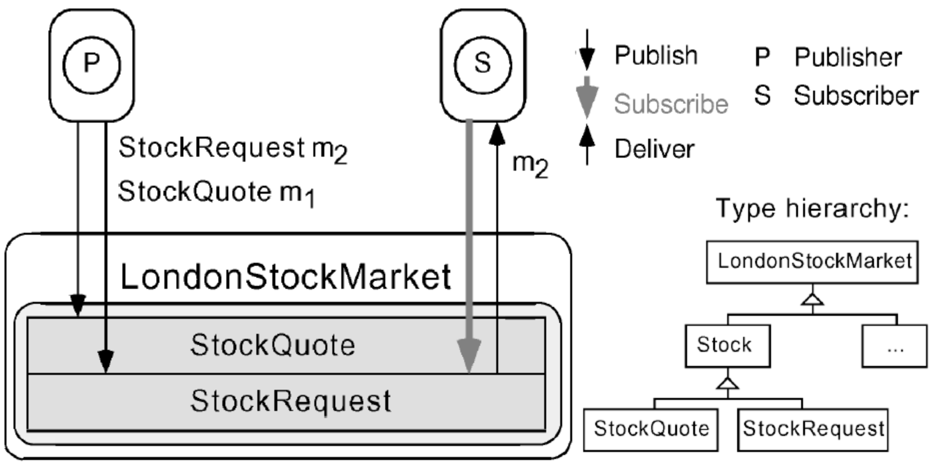
3. Type-based

– Linked with approaches where objects have a type

– Subscriptions defined in terms of <u>types of events</u>

- Can express constraints over attributes or methods of the objects, which allows good expressiveness

↑ Clean integration with OO languages (type safety)



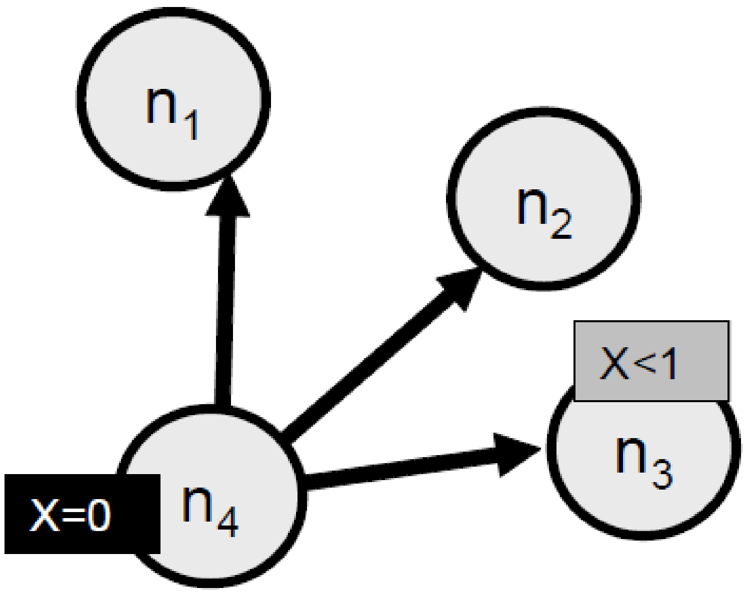

```
public class StockSubscriber implements Subscriber<StockQuote>{
  public void notify(StockQuote s) {
    if (s.getCompany() == 'TELCO' && s.getPrice() < 100)
      buy();
  }
}
// ...
Subscriber<StockQuote> sub = new StockSubscriber();
EventService.subscribe<StockQuote>(sub);
```

Event routing

- Delivering an event to all the subscribers that issued a matching subscription
  1. Flooding (event and subscription flooding)
     - Based on a complete deterministic dissemination of event or subscriptions to the entire system
  2. Selective (filtering- and rendezvous-based)
     - Reduce event dissemination thanks to a deterministic routing structure built upon subscriptions
  3. Event gossiping
     - Probabilistic algorithms with no routing structure, suitable for highly dynamic contexts

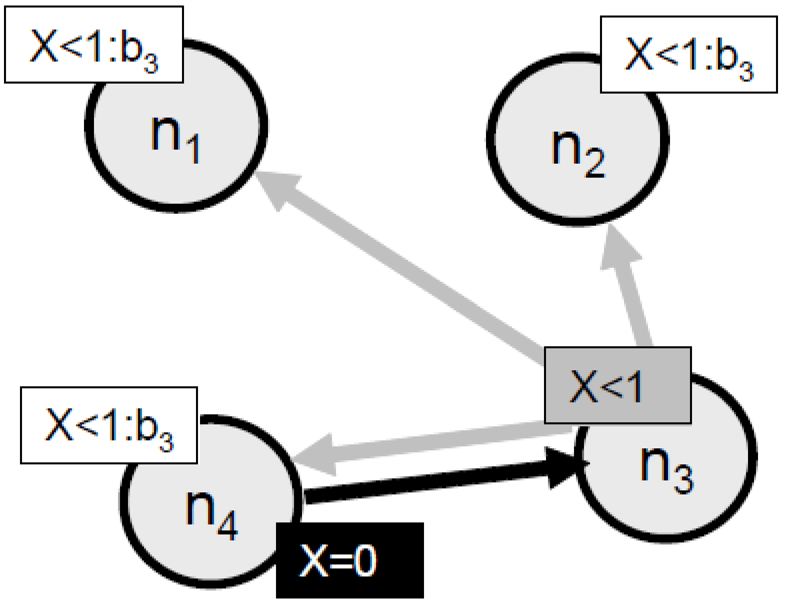
Event routing

1. Event flooding
   – Send events to all nodes in the network
   – Carry out the matching at the subscriber end
   ↓ A lot of unnecessary network traffic
   ↑ Minimal memory overhead at the nodes



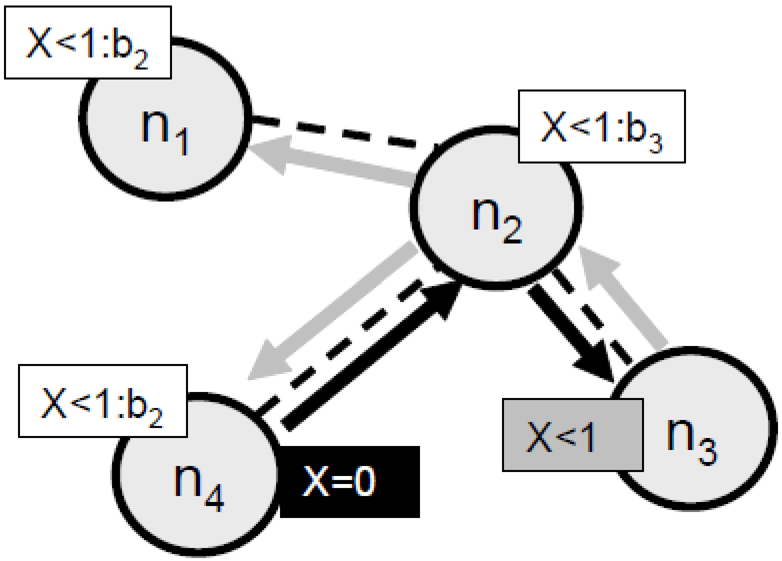
Event routing

2. Subscription flooding
   – Send subscriptions to all nodes
   – Carry out the matching at the publisher end
   – Matched events sent directly to subscribers
   ↓ Network traffic if subscriptions change
   ↓ Memory overhead
   ↑ Fast event notification



Event routing

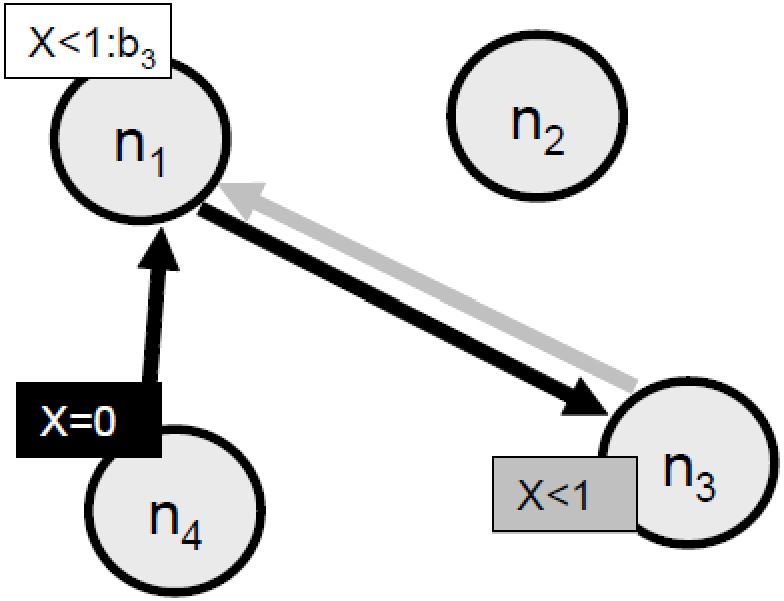
3. Filtering-based
   – Each node stores the set of subscriptions that are reachable through each of its neighbors
   – Send events only to nodes that lay on a path to a valid subscriber
   ↓ Slower event notification
   ↑ Less network traffic and memory use (interaction only with neighbors)



Event routing

4. Rendezvous-based

- Partition subscriptions & events among nodes
  - SN(σ): nodes storing the subscribers of σ
  - EN(e): nodes matching e against subscriptions
  - $e \in \sigma \rightarrow SN(\sigma) \cap EN(e) \neq \emptyset$
  - e.g. use DHTs



- Send event e to nodes $\in$ EN(e)

↓ Rearrange subscriptions when nodes join/leave

↑ Balanced subscription storage and management

Event routing

5. Gossiping
   - Each node chooses **randomly** a few nodes in each round and exchanges events with them
   - Informed gossip: choice can be driven by local information acquired during the node execution (e.g., known subscriptions)

   ↓ Redundancy in message traffic

   ↑ Simple, no memory overhead

   ↑ Supports highly dynamic systems

   ↑ Scales well

Contents

Stream-oriented communication

- In the paradigms presented so far, timing has no effect on correctness
  - **Time-independent** discrete communications
- **Time-dependent** communications
  - Timing is crucial. If wrong, the resulting 'output' from the system will be faulty
  - e.g. audio, video, animation, sensor data
  - a.k.a. 'continuous media' communications
  - Examples:
    - audio: PCM: 1/44100 second intervals on playback
    - video: 30 frames per second (33 ms per image)

Stream-oriented communication

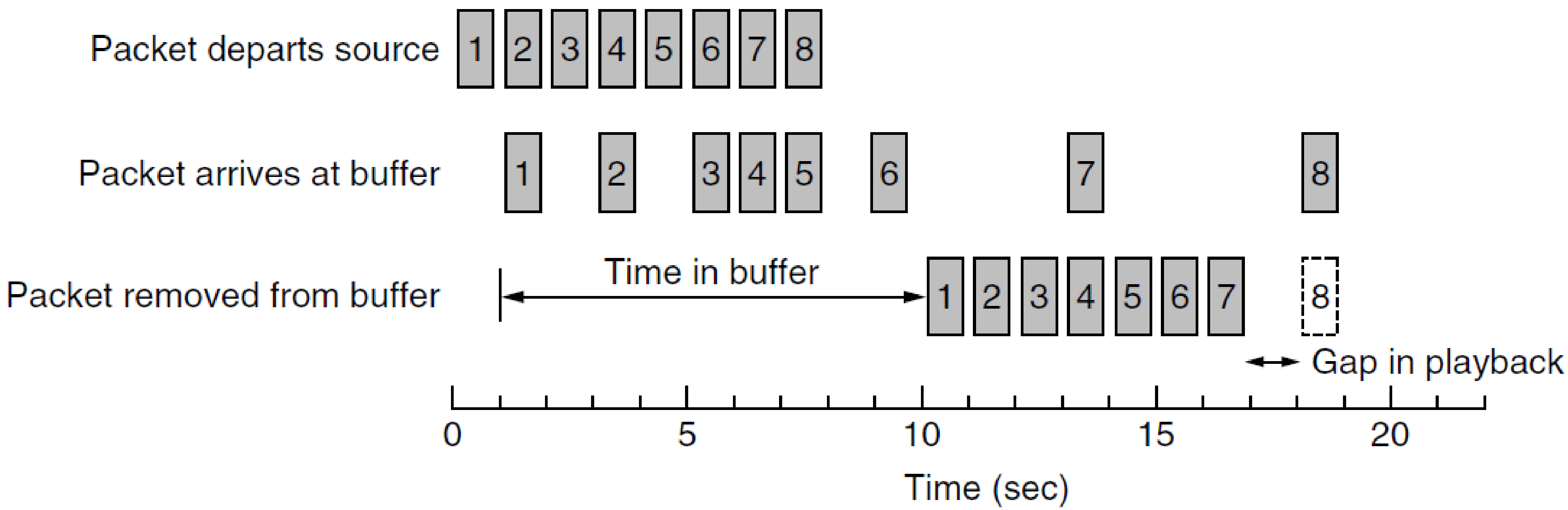
- Stream: unidirectional continuous data flow that supports **isochronous** data transmission
  - There is a maximum and a minimum end-to-end delay ('bounded jitter') for each data unit

A. Simple stream: one single flow of data
   - e.g. audio or video

B. Complex stream: several time-related simple streams (substreams)
   - e.g. stereo audio, combination audio/video
   - Middleware must synchronize delivery of substreams
     - Alternative: multiplex all substreams into a simple stream and demultiplex at the receiver (e.g. MPEG)

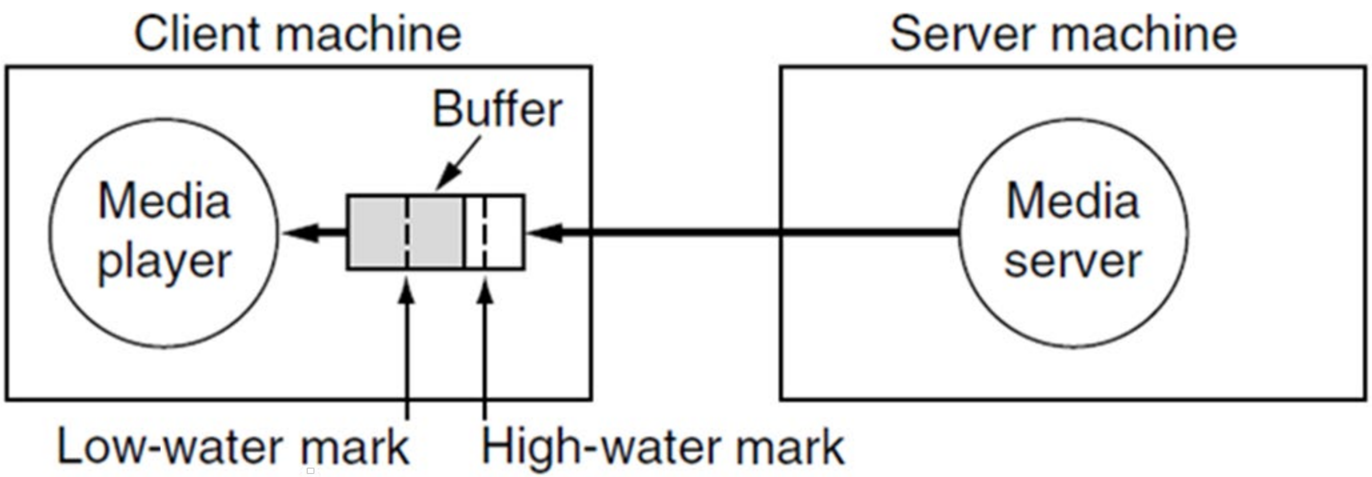
Enforcing timing (QoS)

1. Buffering
   - Mask the end-to-end delay variance (a.k.a. jitter)
   - Buffering allows passing packets to the media application at a regular rate



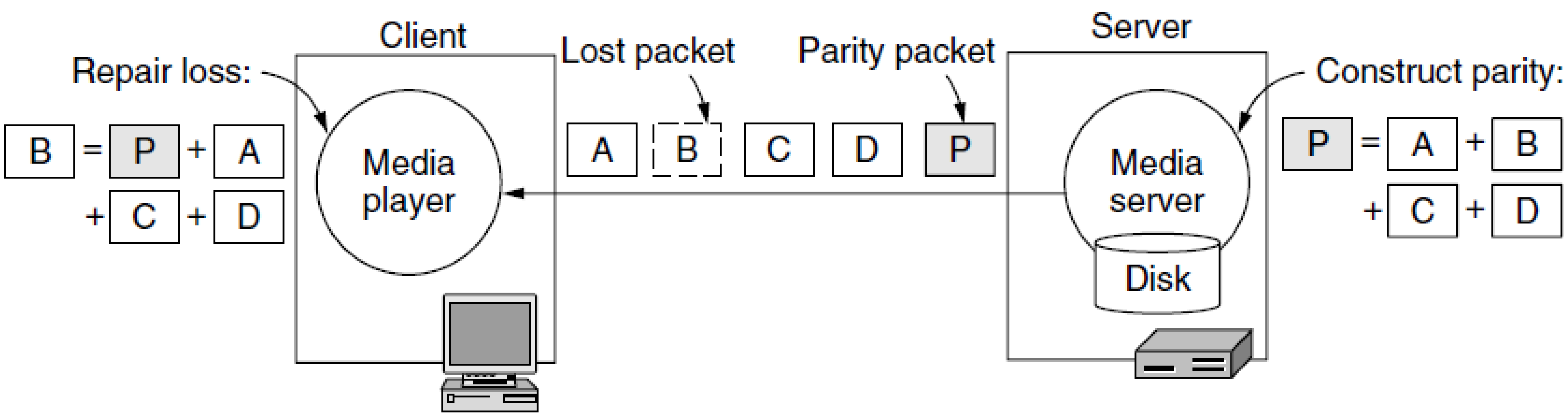
Enforcing timing (QoS)

1. Buffering
   - **Low-water mark** to avoid buffer underrun
     - The client can start to play or will ask the server to resume the transmission (if paused)
   - **High-water mark** to avoid buffer overrun
     - The client will ask the server to pause the transmission



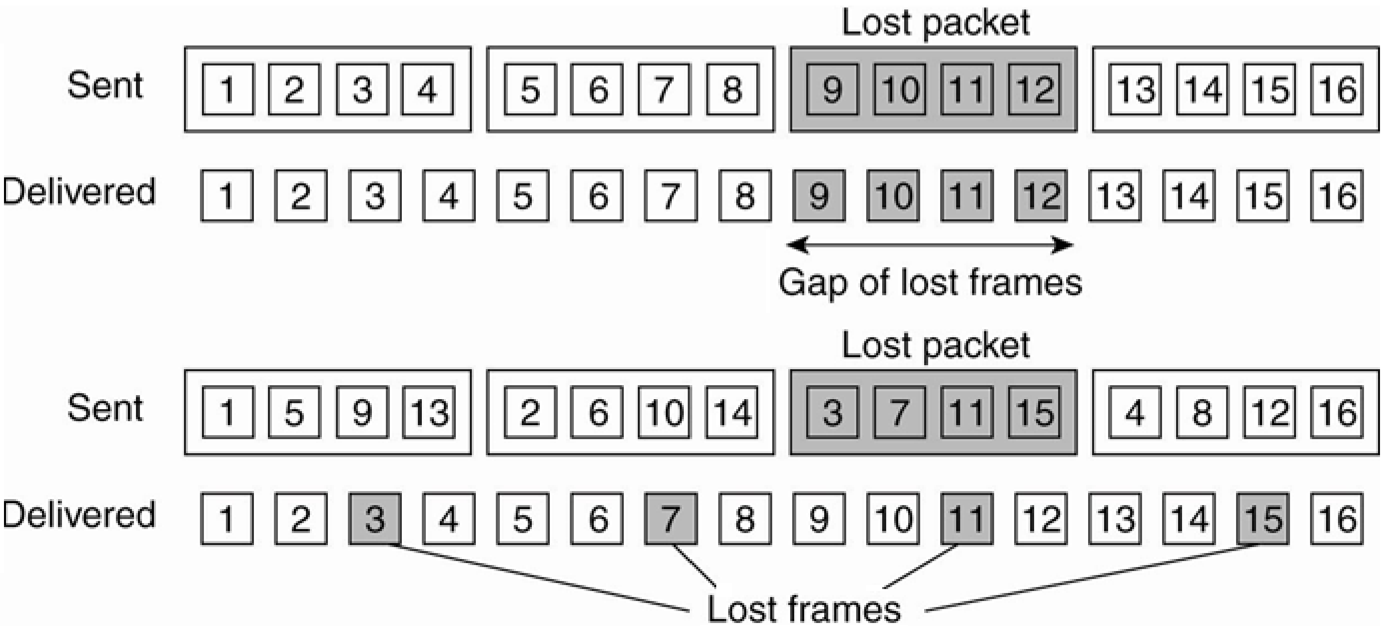
Enforcing timing (QoS)

2. Forward Error Correction (FEC)
   – Retransmission of missing packets is not generally feasible due to timing requirements
   – Send redundant packets that allow to reconstruct the missing ones



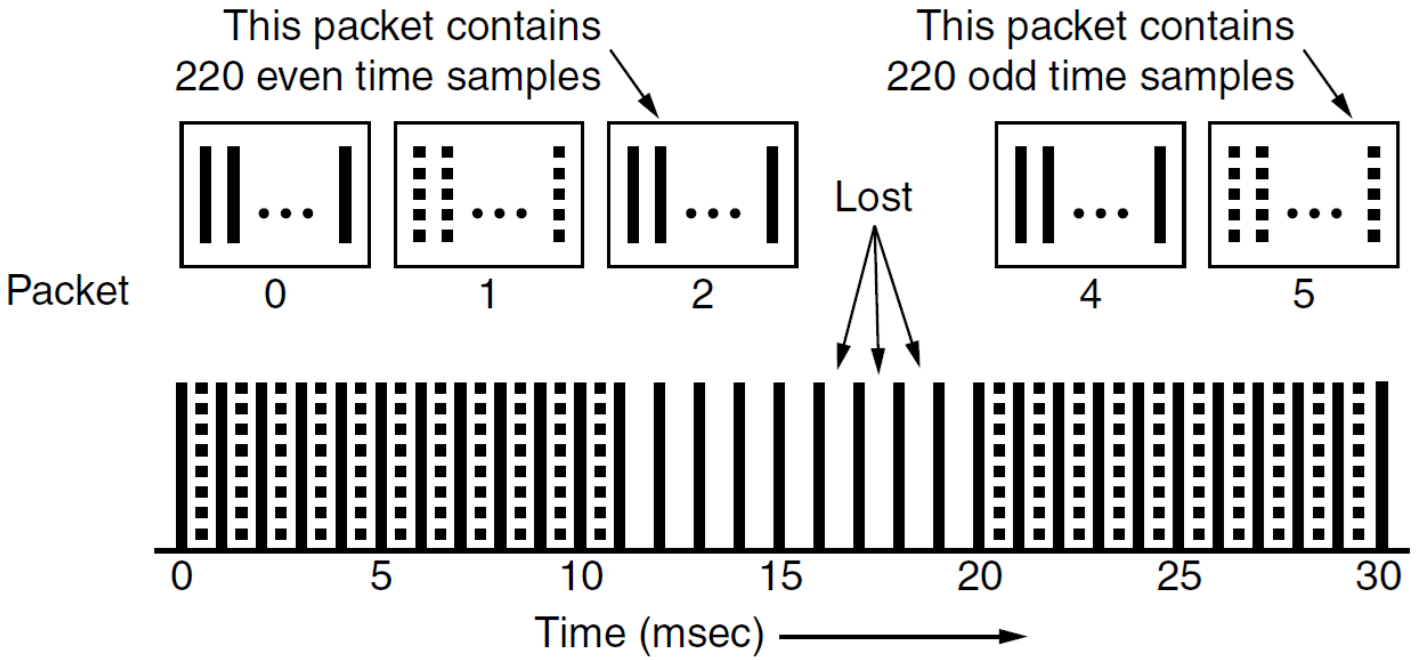
Enforcing timing (QoS)

3. <u>Interleaved transmission</u>
   – Reduce the impact of packet loss
   ↑ Gap is distributed over time
   ↓ Larger buffer required and higher start delay



Enforcing timing (QoS)

3. Interleaved transmission
   - Ex: interleaved (uncompressed) stereo music
     - Instead of a 5 ms gap in the music, we get just lower temporal resolution for 10 ms



Summary

- Powerful and flexible communication is essential
- Network primitives are too low-level
- Middleware communication mechanisms support a higher-level of abstraction
- RPC: synchronous, transient
- Message queues: asynchronous, persistent
- Publish-subscribe: decoupled in space
- Streams: for 'temporally-related data'
- Further details:
  - [Tanenbaum]: chapters 4, 5.6, and 8.3
  - [Coulouris]: chapters 4, 5, 6, and 20