2. Interprocess Communication

Sistemes Distribuïts en Xarxa (SDX)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2018/2019 Q2

Contents

- **Introduction**
- Remote invocation
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

Interprocess communication

- The ‘core’ of every distributed system
- Communication is always based on **low-level message passing** offered by the underlying network
  - Established network facilities are too primitive
  - Systems are too difficult to develop
- Communicating processes must agree on a set of rules $\Rightarrow$ **protocols**
- Format messages conform to protocols

Contents

- Introduction
  - **Types of communication**
  - Communication paradigms
- Remote invocation
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

Types of communications

A. Direct communication
   – Senders explicitly direct messages/invocations to the associated receivers
   – Senders must know receivers identity and both must exist at same time
   – e.g. sockets, remote method invocations

B. Indirect communication
   – 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 communications

A. Space decoupling
   - Senders do not need to know who they are sending to
   - e.g. event-based systems (publish-subscribe)

B. Time decoupling
   - The sender and the receiver do not need to exist at the same time
   - e.g. e-mail
   - Time decoupling also allows distinguishing persistent from transient communications

Types of communications

A. Persistent communications
   - The receiver does not need to be operational at the communication time
     - The message is **stored** at a communication server as long as it takes to deliver it at the receiver
   - e.g. e-mail

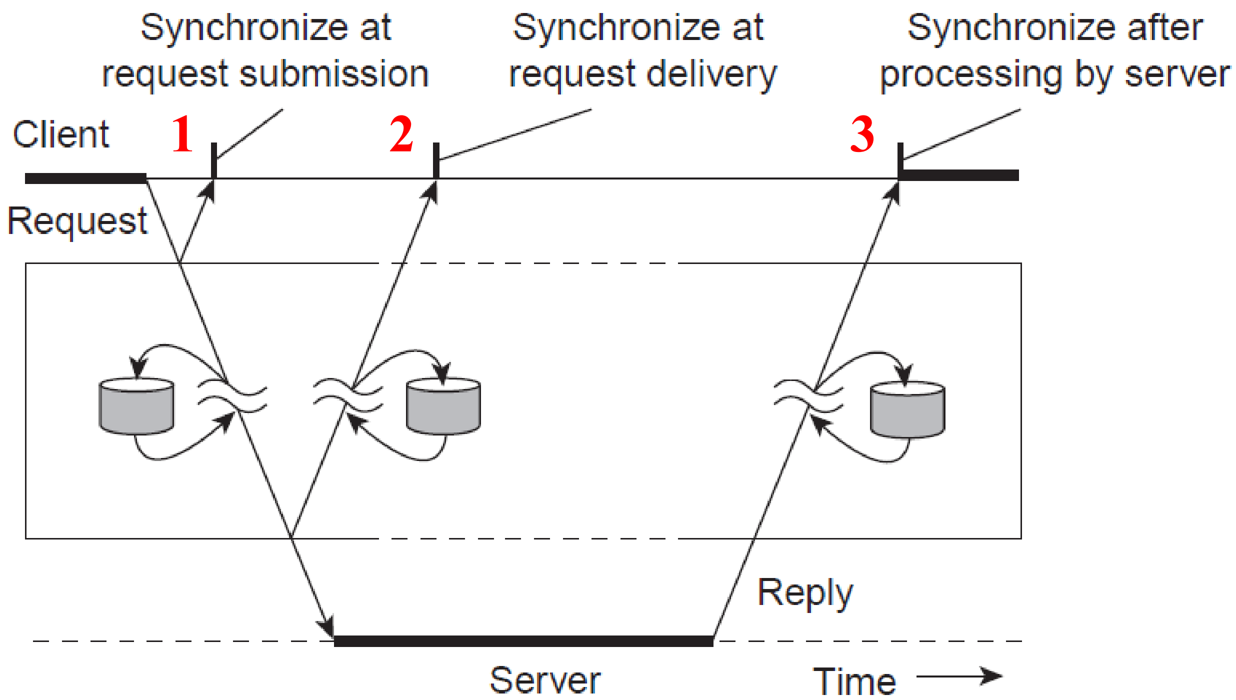
B. Transient communications
   - A message is **discarded** by a communication server as soon as it cannot be delivered at the next server, or at the receiver
   - e.g. sockets, remote method invocations

Types of communications

A. Asynchronous communications
   - The sender **continues** with other work immediately upon sending a message to the receiver
   - e.g. publish-subscribe systems, e-mail

B. Synchronous communications
   - The sender **blocks, waiting** for a reply from the receiver, before doing any other work
   - This tends to be the default model for request-reply paradigms (e.g. RPC/RMI)

Synchronous communications



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 communications

A. Discrete communications
   - Exchange of ‘independent’ units of information
   - Timing has no effect on correctness
   - e.g. e-mail, remote method invocations

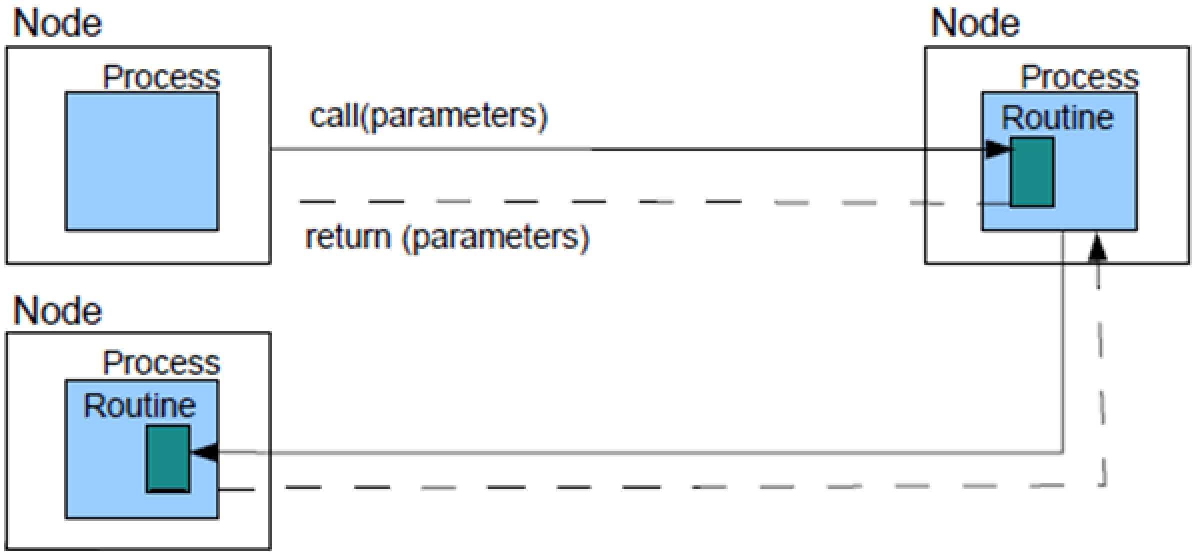
B. Continuous communications
   - Messages are related to each other by the order they are sent, or by a temporal relationship
   - Timing between data items must be preserved to interpret correctly the data
   - e.g. streams (audio, video, …)

Contents

- **Introduction**
  - Types of communication
  - **Communication paradigms**
- Remote invocation
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

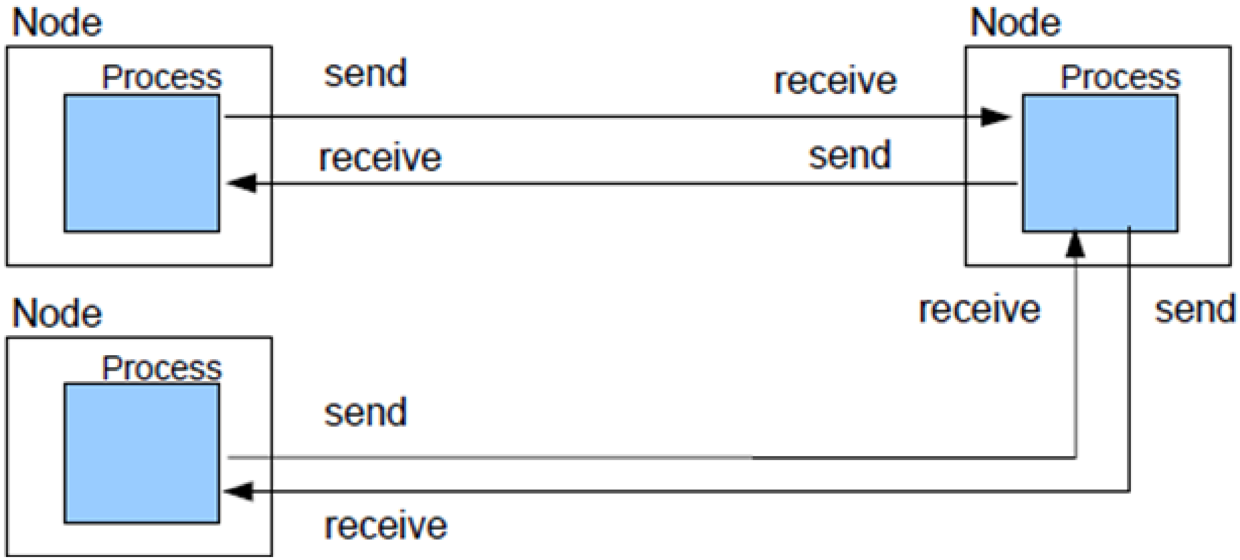
Remote invocation

- Transparent extension to traditional programming: a node can call a function in another one as if it was local (e.g. RPC, RMI)
- Direct, transient, synchronous point-to-point interactions
- Middleware handles the marshaling/unmarshaling of parameters
- Protocol is sessionless and server is stateless about client
  - Function can change server's state, but client must maintain its state



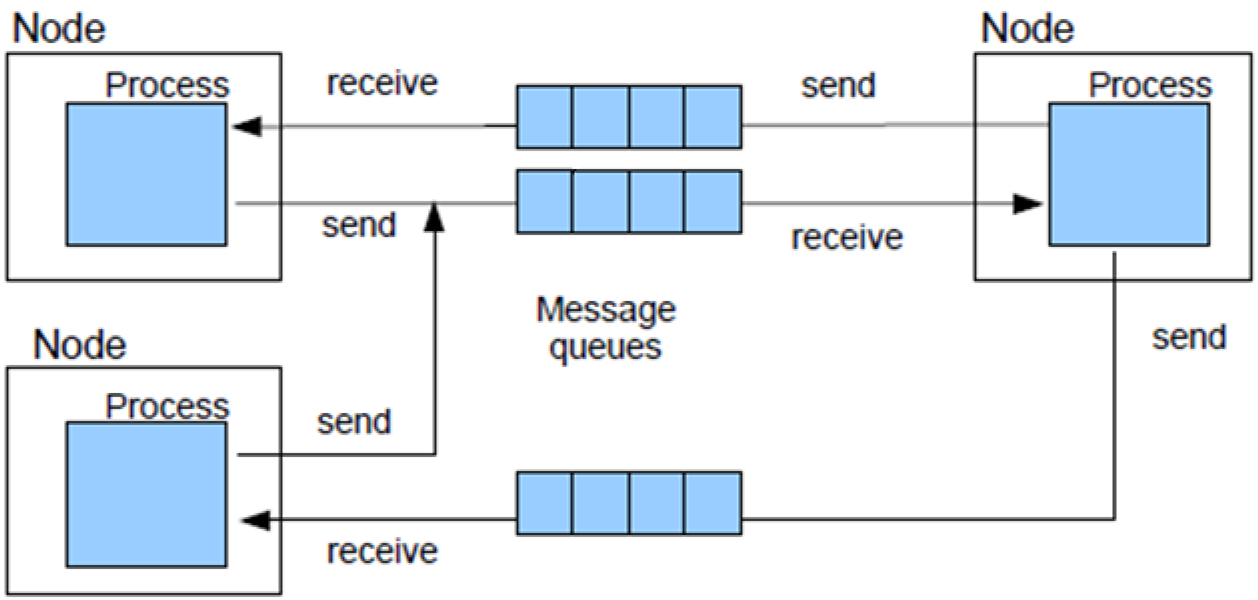
Message passing

- <u>Direct</u>, <u>transient</u> networked communication between processes
  - e.g. sockets, MPI
- Generally <u>synchronous</u> and point-to-point
  - MPI supports also non-blocking and multipoint communication
- It is not middleware mediated
- Lacks transparency: exposes the network characteristics/issues



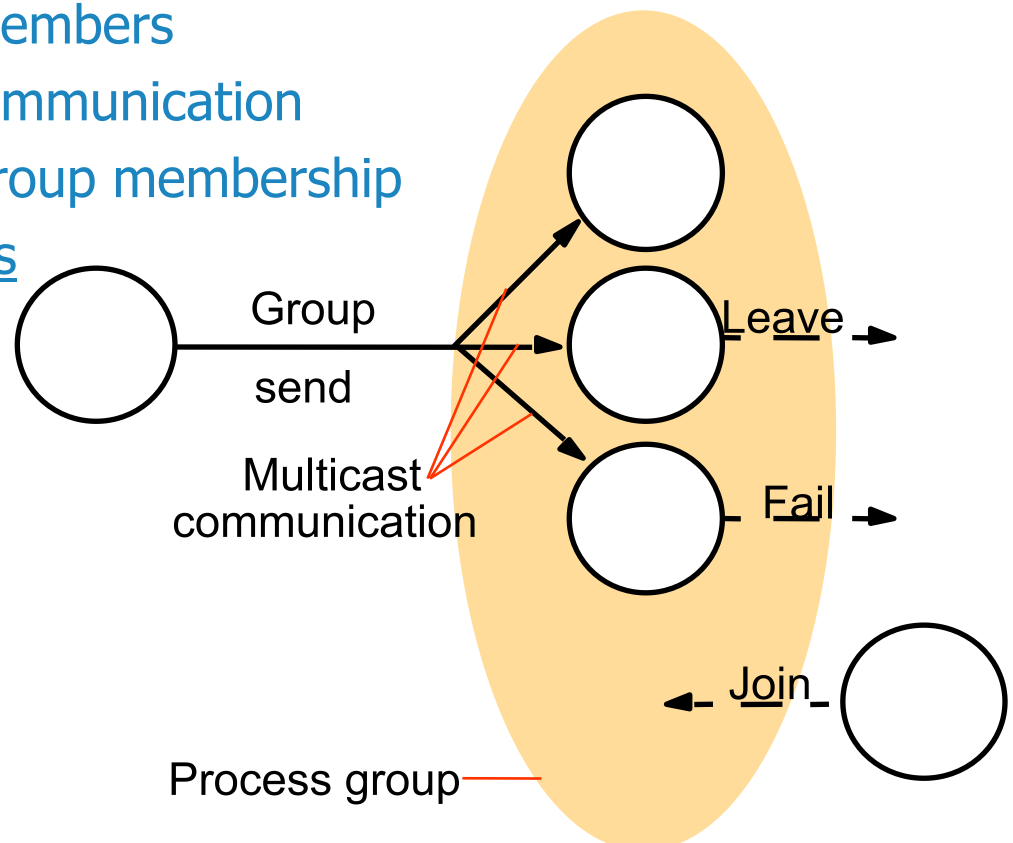
Message queuing

- Sender puts message into a queue, receiver gets it from queue
  - e.g. e-mail, Message Oriented Middleware (Apache ActiveMQ*, RabbitMQ*)
- <u>Persistent</u> and <u>asynchronous</u>, by means of message queues
- Point-to-point (sender ⇔ queue ⇔ receiver)
- Middleware stores/forwards messages



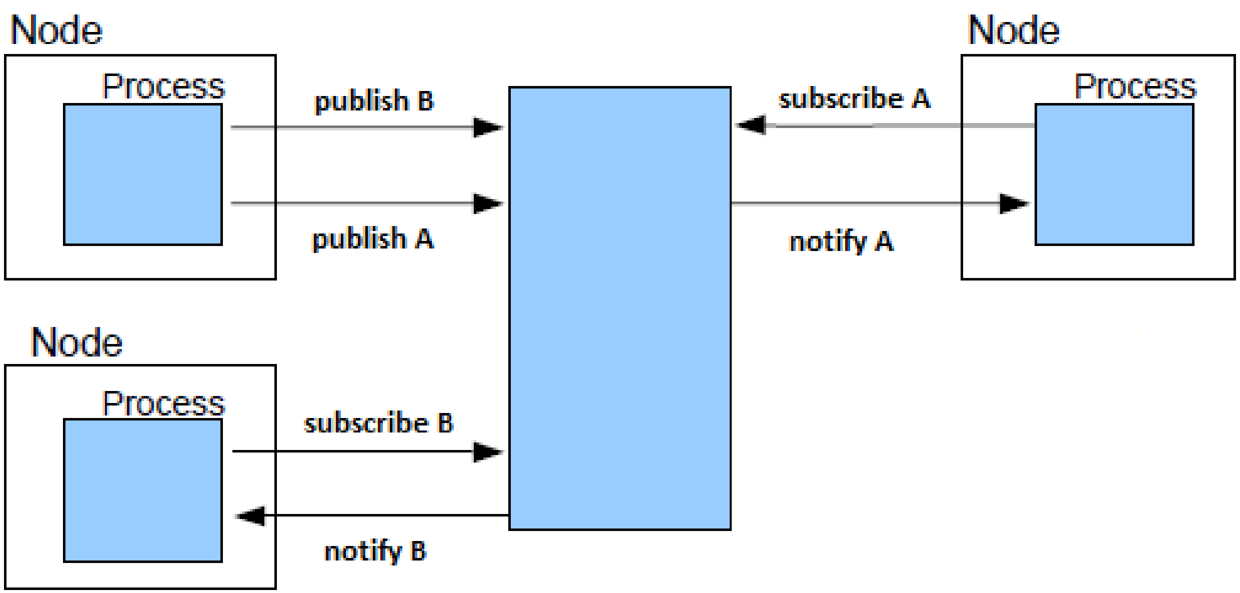
Group communication

- 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
- One-to-many style of communication
- Middleware maintains group membership
- Transient & synchronous
- e.g. JGroups, Spread



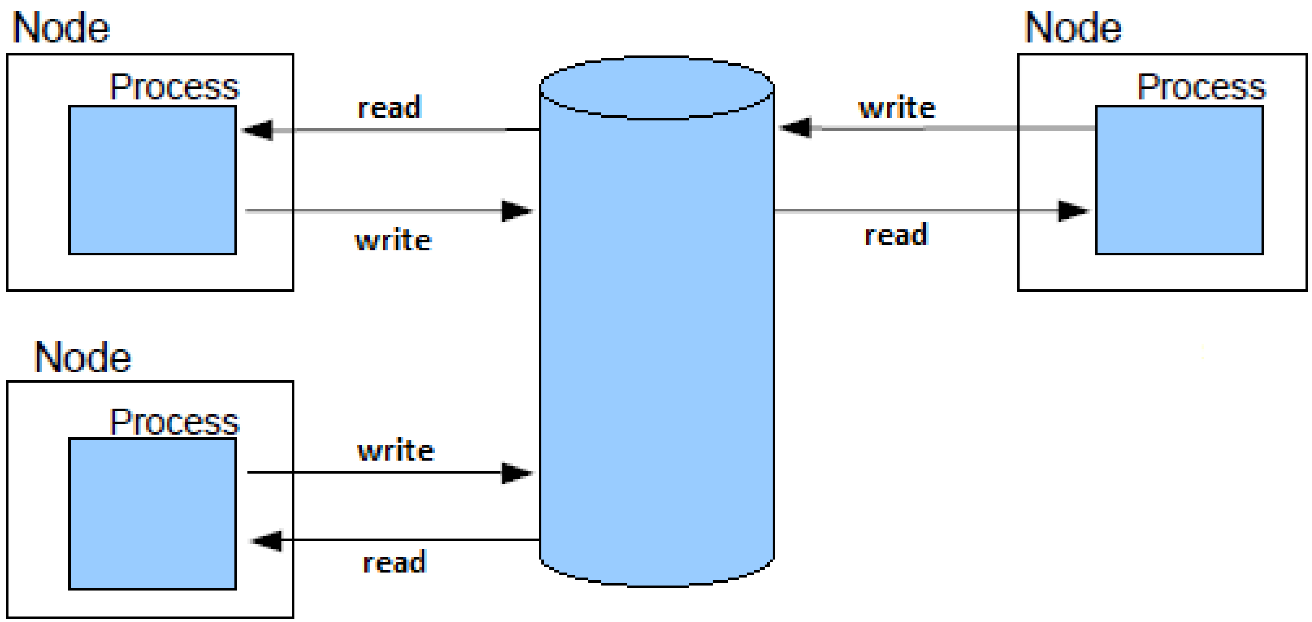
Publish-subscribe

- Indirect, asynchronous communication by propagating events
  - Producers publish structured events, consumers express interest in events through subscriptions (e.g. Apache Kafka, Apache ActiveMQ*, Scribe)
- One-to-many style of communication (space decoupled)
- Middleware efficiently matches subscriptions against published events and ensures the correct delivery of event notifications



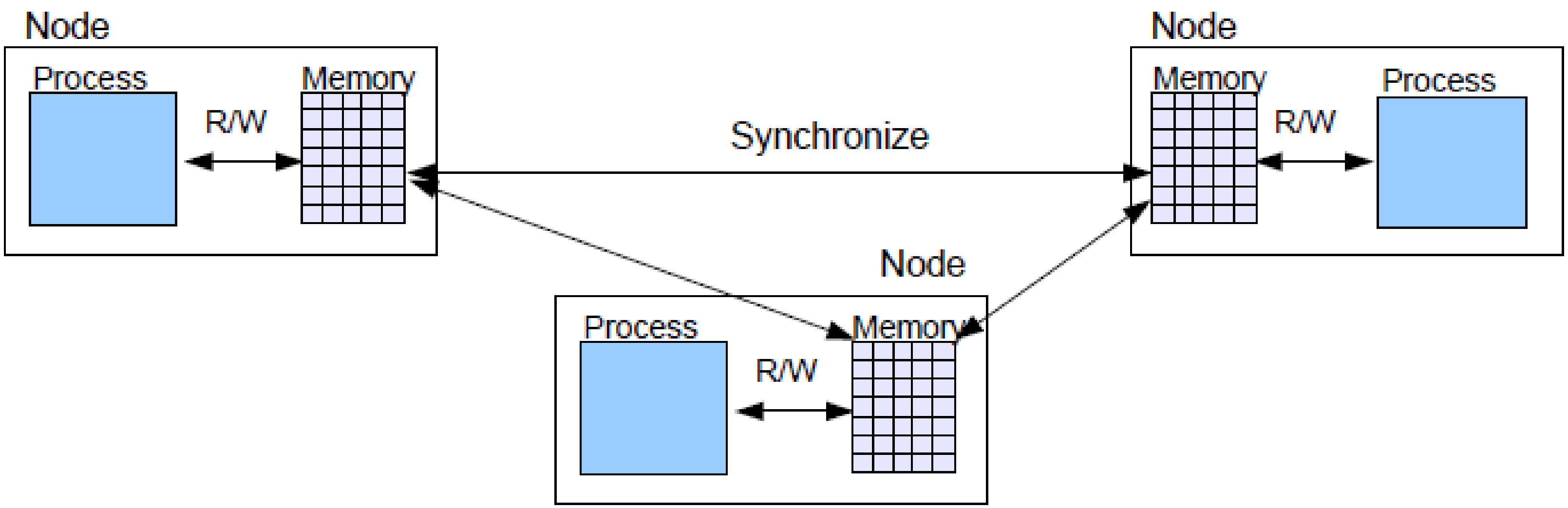
Shared data space

- Persistent, asynchronous communication using a shared storage
  - e.g. JavaSpaces, which also provides space decoupling by means of pattern matching on contents
- Post items to shared space; consumers pick up at a later time
- One-to-many style of communication



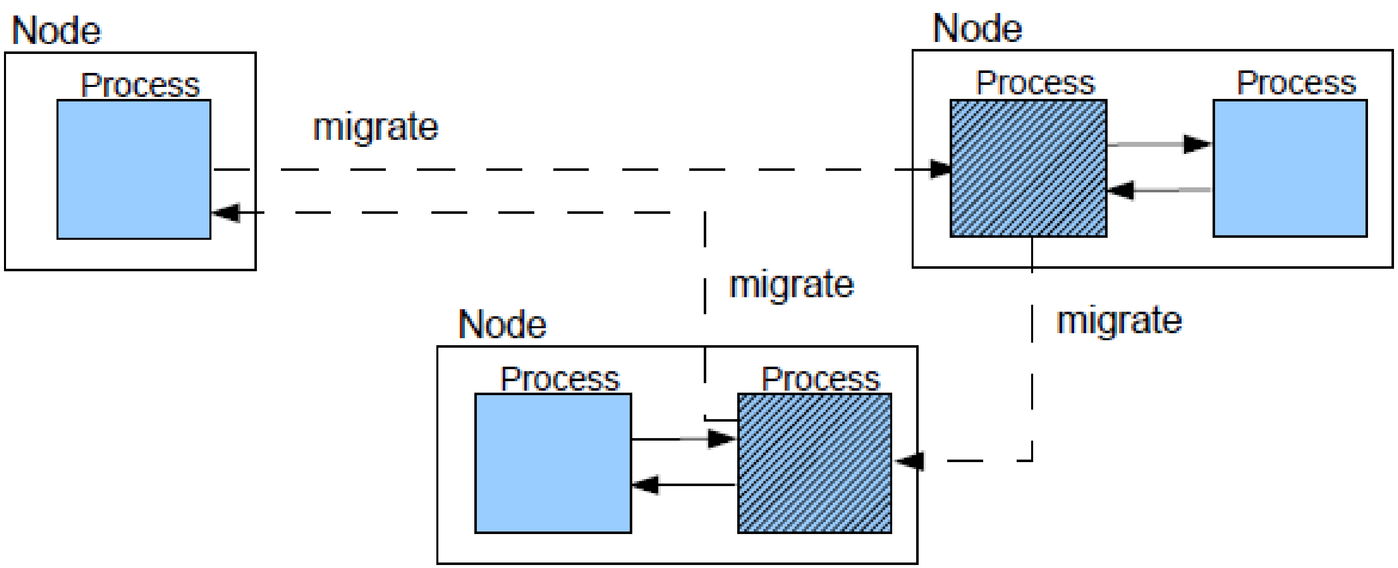
Shared memory

- 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
- <u>Indirect</u> (space decoupled) and <u>transient</u> communication
- Interaction is multipoint (many components share memory)
- Middleware synchronizes and maintains the consistency of data



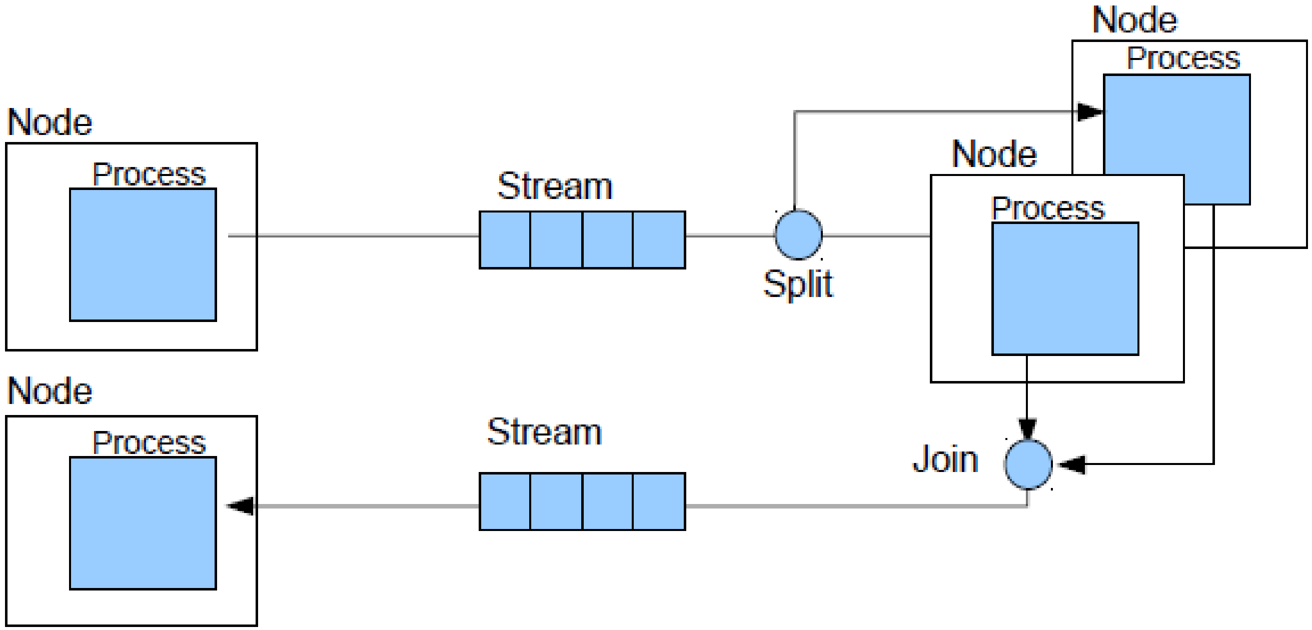
Mobile code / agents

- 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 it is a potential security threat
- Middleware transfers code and saves/restores process state



Stream oriented

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



Contents

- Introduction
- **Remote invocation**
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

Contents

- Introduction
- **Remote invocation**
  - **Remote Procedure Call (RPC)**
    - Basic RPC operation
    - RPC parameter passing
    - Extended RPC models
    - RPC semantics and failures
  - Remote Method Invocation (RMI)
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

Remote Procedure Call (RPC)

- Call procedures located on other machines
  - Hide communication between caller & callee by using procedure-call mechanism
    - Programmers do not have to worry about all 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

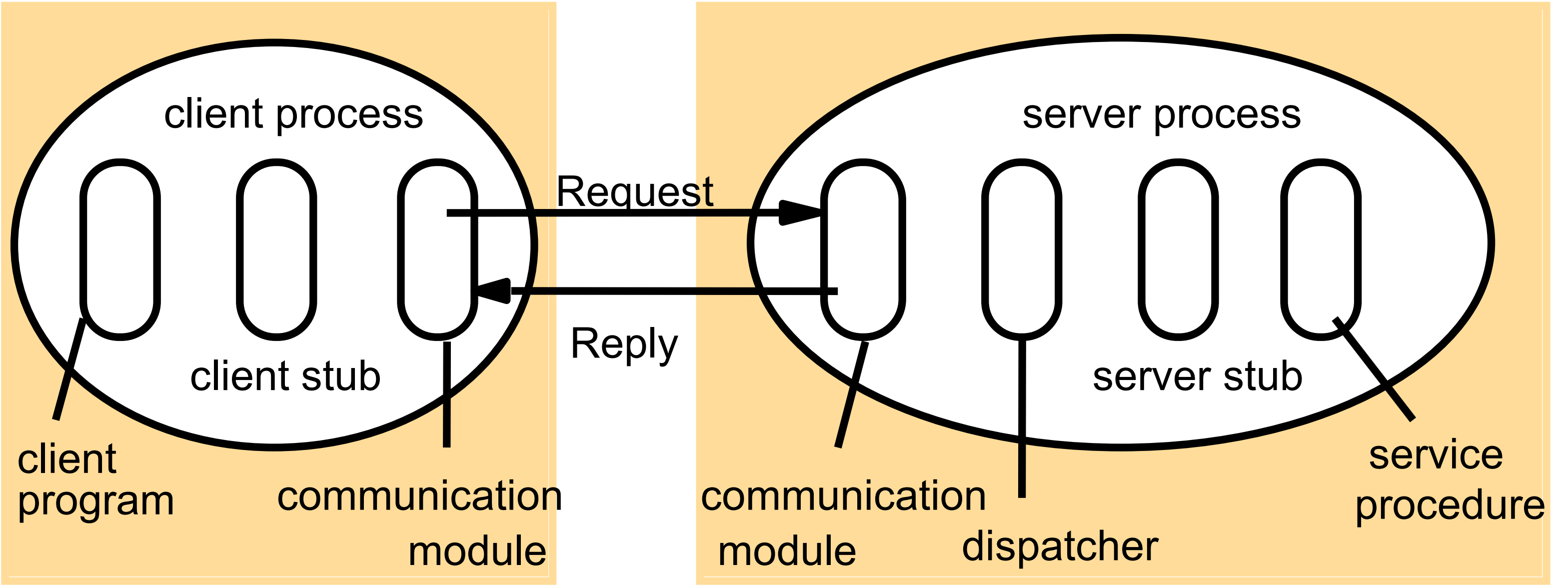
- Introduction
- **Remote invocation**
  - **Remote Procedure Call (RPC)**
    - **Basic RPC operation**
    - RPC parameter passing
    - Extended RPC models
    - RPC semantics and failures
  - Remote Method Invocation (RMI)
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

Basic RPC operation

- RPCs are **transparent**
  - From the programmer viewpoint, a ‘remote’ procedure call looks and works identically to a ‘local’ procedure call
- Transparency is achieved by using **stubs**:
  - The **CLIENT stub**
    - Implements the interface on the local machine through which the remote functionality can be invoked
  - The **SERVER stub**
    - Transforms requests coming in over the network into local procedure calls

Basic RPC operation

- The steps of a RPC

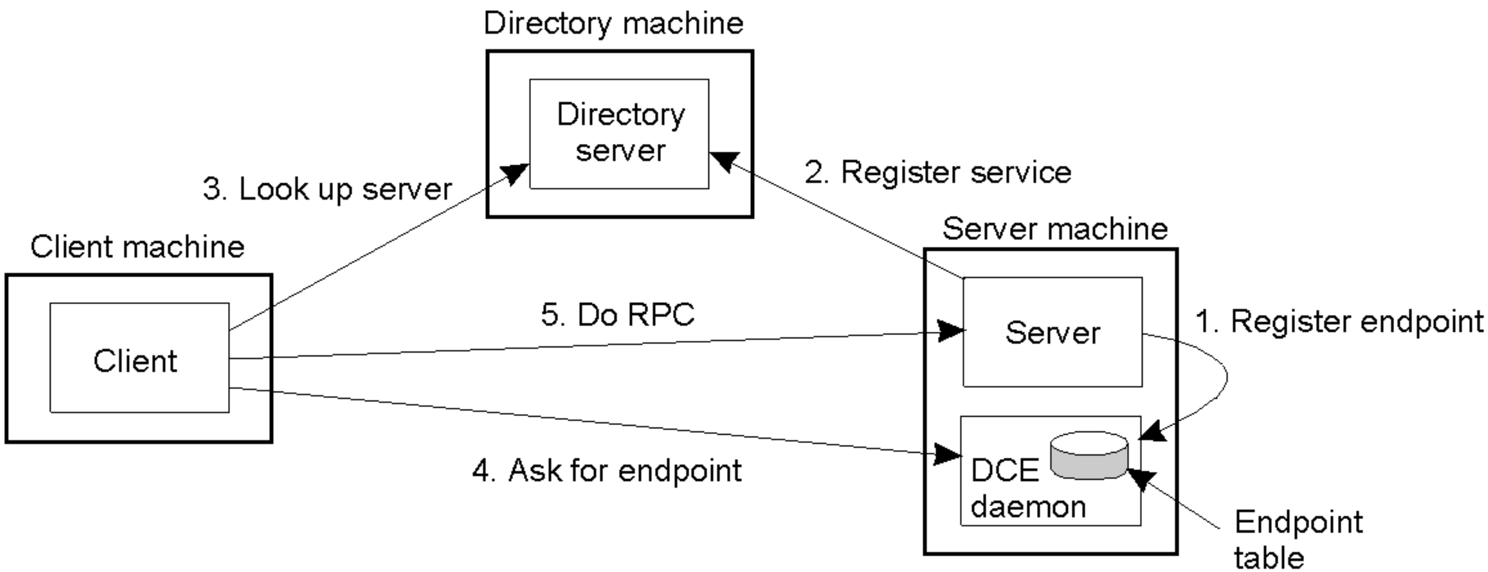


Basic RPC operation

- The steps of a RPC
  1. Client procedure calls client stub in normal way
  2. Client stub builds message, calls local OS
  3. Client's OS sends message to remote OS
  4. Remote OS gives message to server stub
  5. Server stub unpacks parameters, calls server
  6. Server does work, returns result to the stub
  7. Server stub packs it in message, calls local OS
  8. Server's OS sends message to client's OS
  9. Client's OS gives message to client stub
  10. Stub unpacks result, returns to client

Basic RPC operation

- Client-to-server binding
  - Locate server machine, and then locate the server
    - Use a ‘**directory service**’ $\Rightarrow$ binder
- e.g. DCE RPC

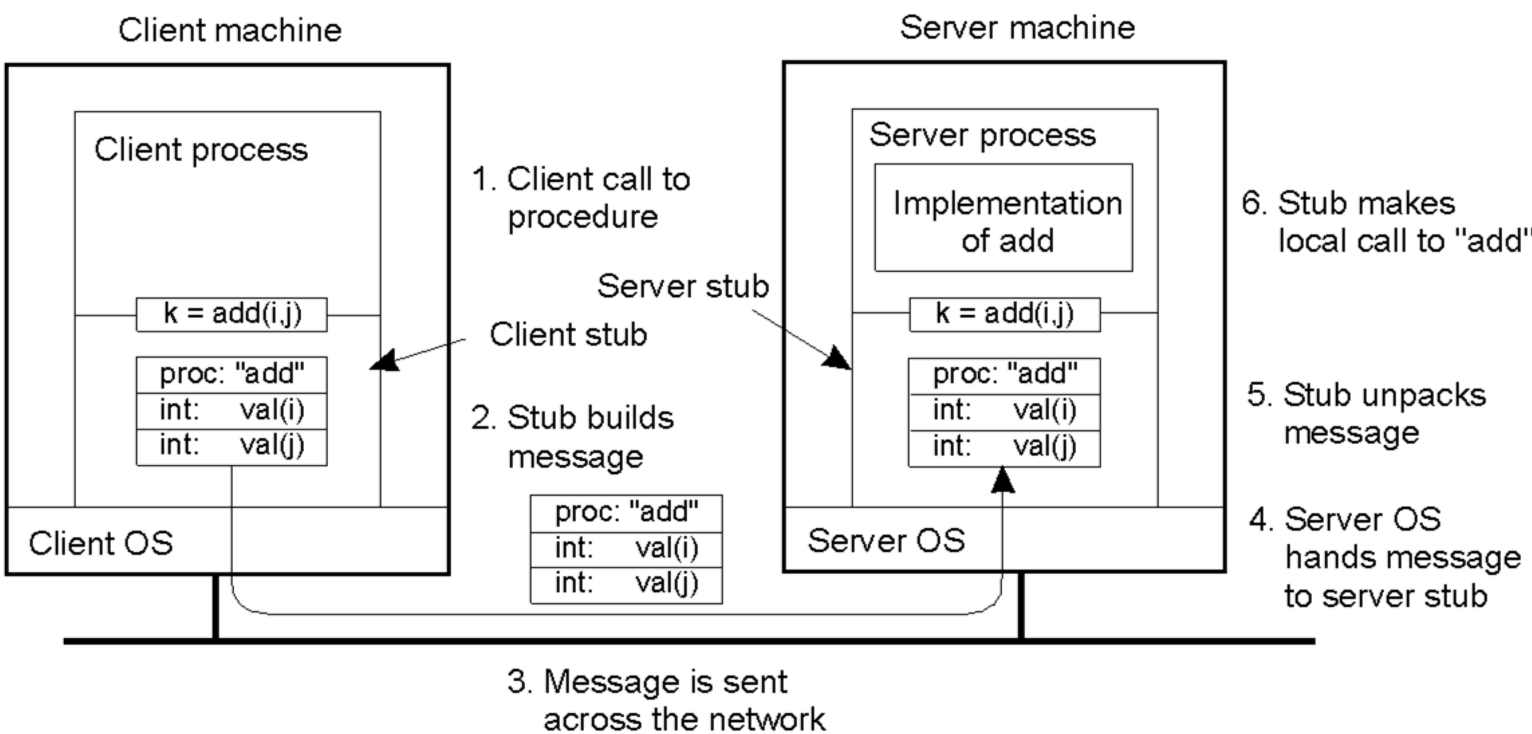


Contents

- Introduction
- **Remote invocation**
  - **Remote Procedure Call (RPC)**
    - Basic RPC operation
    - **RPC parameter passing**
    - Extended RPC models
    - RPC semantics and failures
  - Remote Method Invocation (RMI)
- 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
      - IBM mainframes: EBCDIC, IBM PC: ASCII
    - use different byte-ordering
      - Intel: little endian, Sun SPARC: big endian
  - Solution: Agree on the protocol used
    - Representation of data, format and exchange of messages, etc.
    - Use a standard representation
      - Example: SUN eXternal Data Representation (XDR)

RPC parameter passing

- Passing reference parameters
  - Pointers are meaningful only within a specific address space
  - By default, RPC does not offer call by reference
  - Some implementations allow passing by reference **arrays (of known length) & structures**
    - Copy/restore semantics
      - Pass a copy and the server stub passes a pointer to the local copy
    - IN/OUT/INOUT markers
      - May eliminate one copy operation

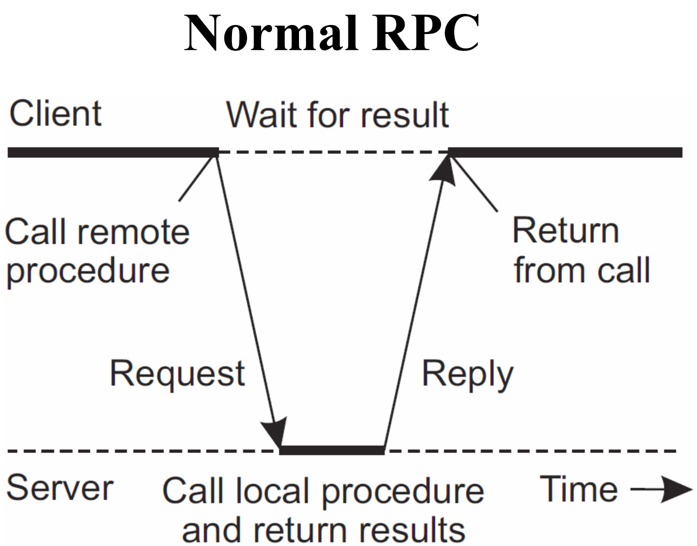
Interface Definition Language (IDL)

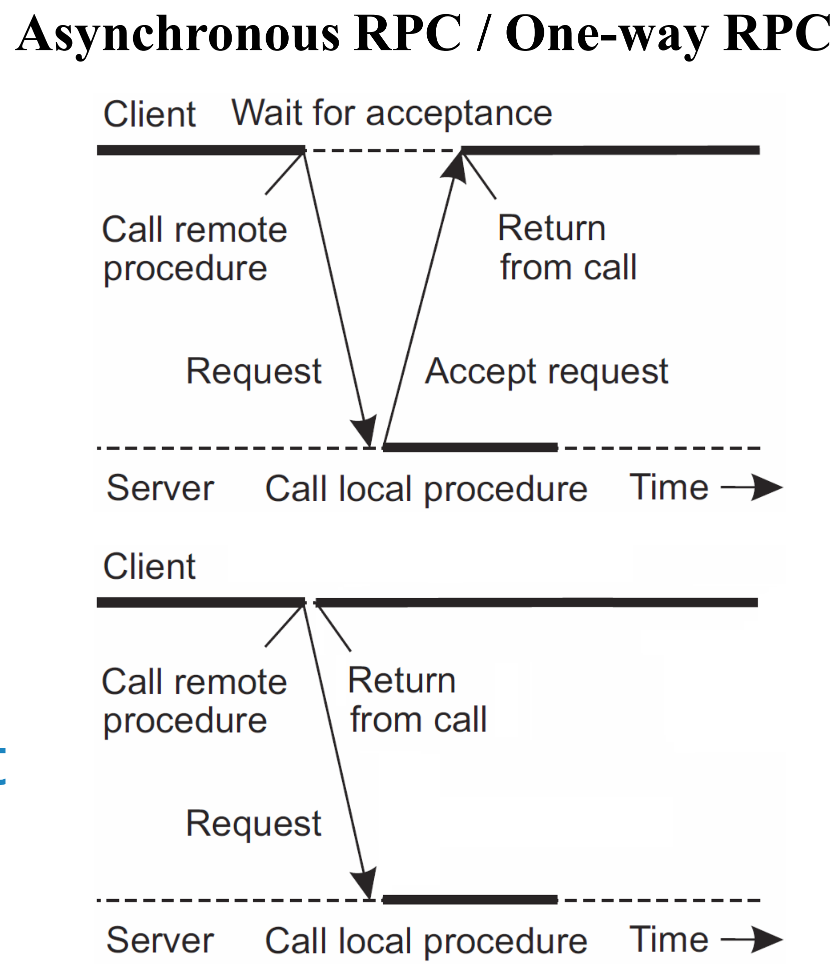
- Definition of interfaces simplifies RPCs
  - Service interface specifies the procedures offered by a server, defining the types of the arguments of each of the procedures
- Interfaces are defined by means of an IDL
  - e.g. XDR (SUN RPC); WSDL (Web Services)
  - IDLs are language-neutral
    - Don't 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 invocation**
  - **Remote Procedure Call (RPC)**
    - Basic RPC operation
    - RPC parameter passing
    - **Extended RPC models**
    - RPC semantics and failures
  - Remote Method Invocation (RMI)
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

Extended RPC models: Asynchronous RPC

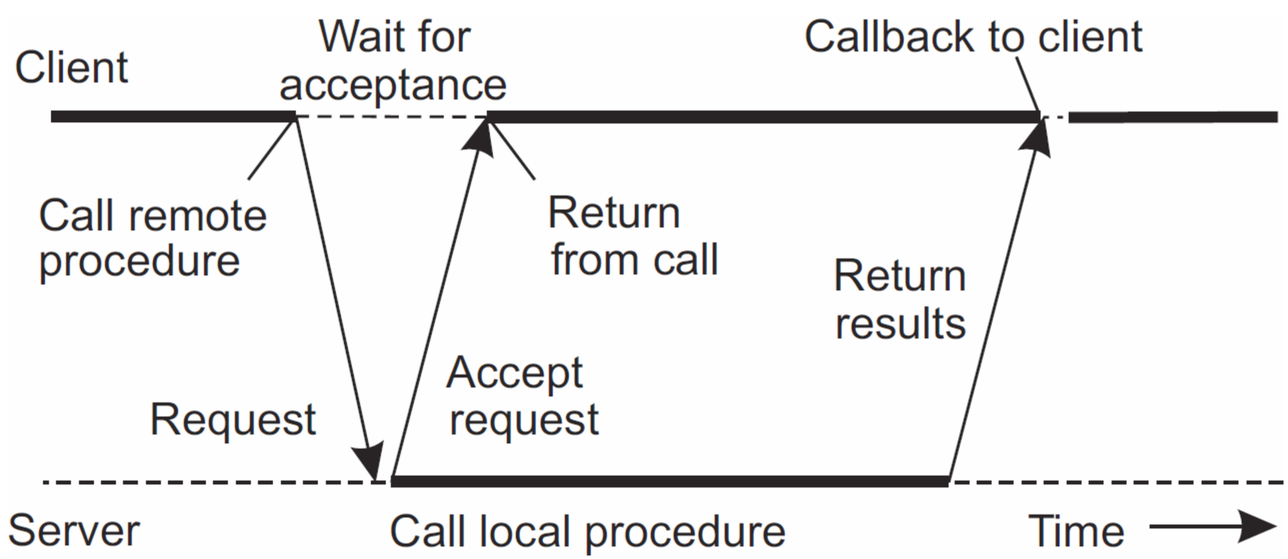




- Useful when the client does not need or expect a result

Extended RPC models: Deferred Synchronous RPC

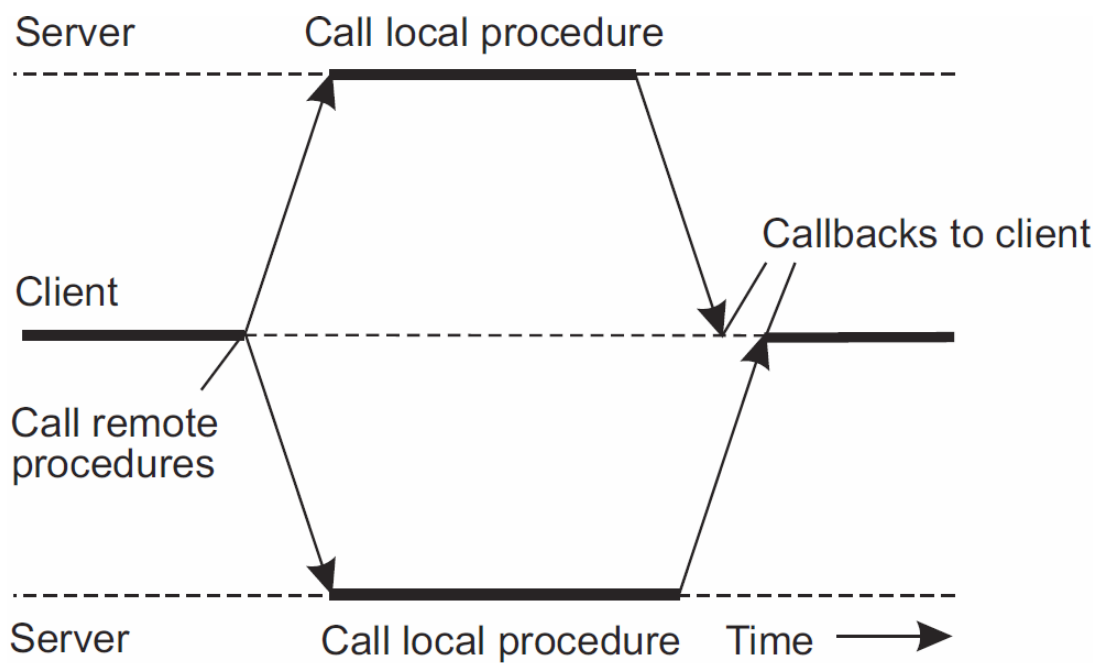
- Communication through an asynchronous RPC and a **callback**



↑ 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

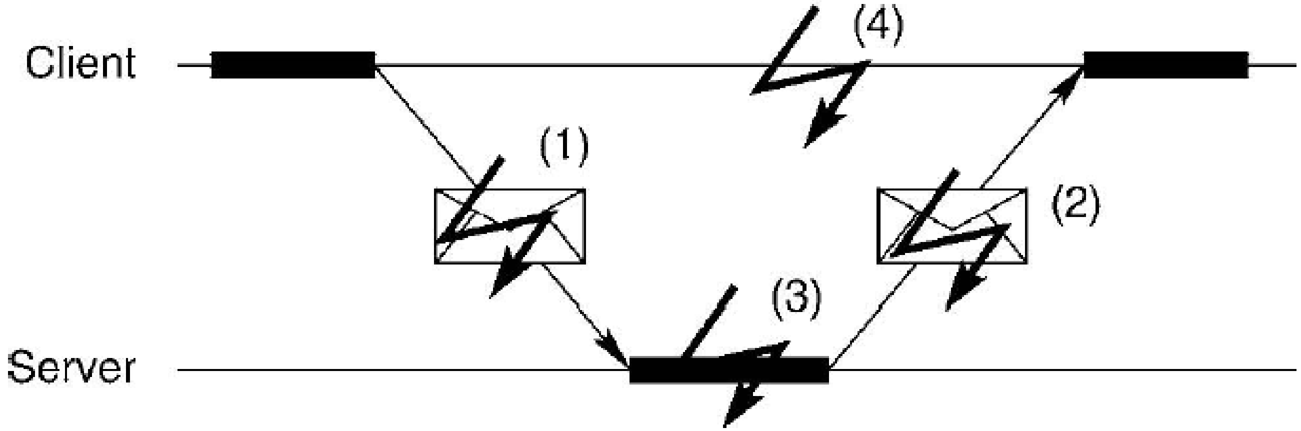


Contents

- Introduction
- **Remote invocation**
  - **Remote Procedure Call (RPC)**
    - Basic RPC operation
    - RPC parameter passing
    - Extended RPC models
    - **RPC semantics and failures**
  - Remote Method Invocation (RMI)
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

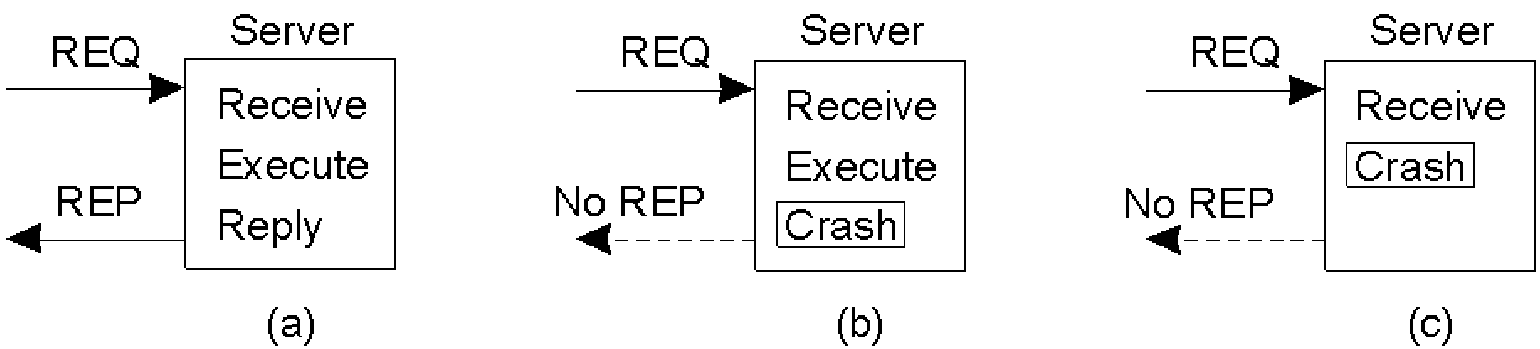
RPC semantics and failures

- 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 $\Rightarrow$ 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
       - History must 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 <u>at least once, but possibly more</u>, in case of lost request or reply messages
- RPC may be executed <u>several times, or possibly not at all</u>, 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
    - <u>Pure read operations</u>: e.g. loading a static web page
    - <u>Strict overwrite operations</u>: e.g. update your billing address in an online shop

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

c) At-most-once semantics

- RPC will be executed <u>exactly once</u> in case of lost request or reply messages
- RPC will be executed <u>at most once, or possibly not at all</u>, 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 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 → 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

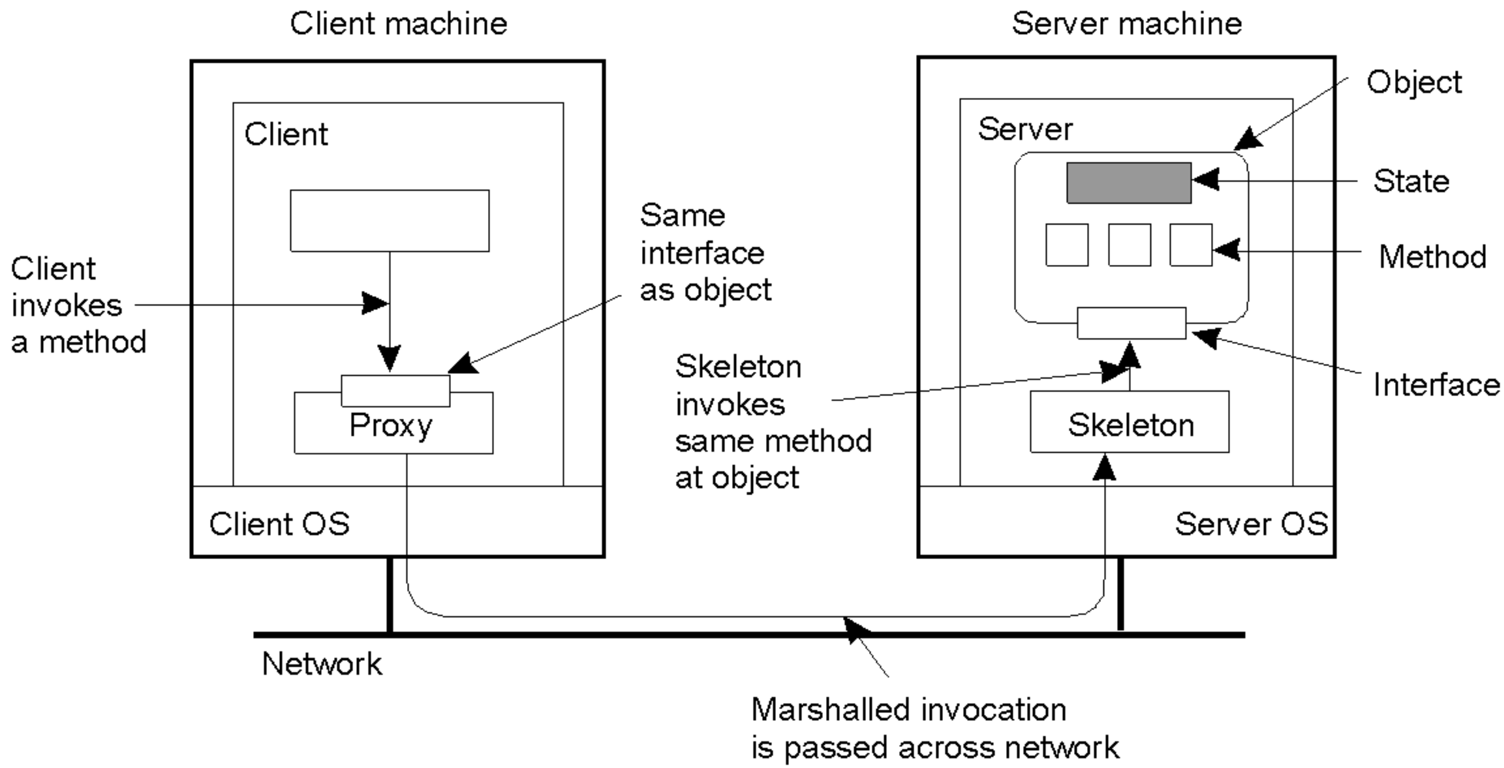
Contents

- Introduction
- **Remote invocation**
  - Remote Procedure Call (RPC)
    - Basic RPC operation
    - RPC parameter passing
    - Extended RPC models
    - RPC semantics and failures
  - **Remote Method Invocation (RMI)**
- Message-oriented communication
- Event-based communication
- Stream-oriented communication

Remote Method Invocation (RMI)

- Idea: expansion of the RPC mechanism to support object oriented systems
  - e.g. Java RMI
- Distributed objects
  - Object state is stored on the server and can be accessed only by the methods of the object
  - Every remote object has an interface that specifies which of its methods can be invoked remotely
  - Remote objects can receive remote invocations if we have its remote object reference
  - Tighter integration than RPC into the language

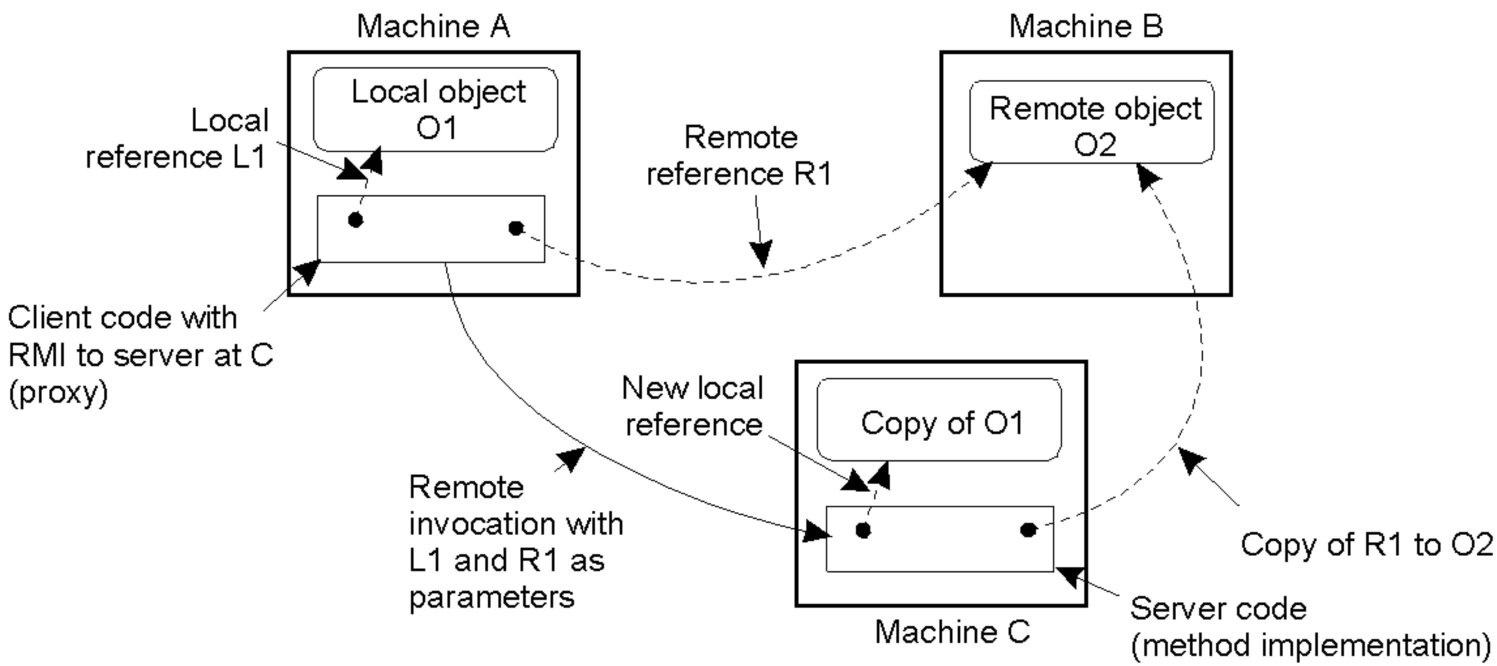
Remote Method Invocation (RMI)



- Distributed object
  - The **proxy** can be thought of as the 'client stub'
  - The **skeleton** can be thought of as the 'server stub'

RMI: Parameter passing

- Less restrictive than RPC
  - Supports **system-wide** object references



  - Local objects (O1) by value; Remote ones (O2) by reference

RMI: Java example

```java
import java.rmi.*;

public interface Hello extends Remote {
  String sayHello() throws RemoteException;
}
```

REMOTE INTERFACE

```java
import java.rmi.*;

public class Client {
  public static void main(String args[]) {
    try {
      Registry registry = LocateRegistry.getRegistry(args[0]);
      Hello stub = (Hello) registry.lookup("Hello");
      String resp = stub.sayHello();
      System.out.println("response: " + resp);
    } catch (Exception e) { ... }
  }
}
```

CLIENT PROGRAM

RMI: Java example

```
import java.rmi.*;

public class Server implements Hello {
  public String sayHello() {
    return "Hello, world!";
  }
  public static void main(String[] args) {
    try {
      Server obj = new Server();
      Hello stub = (Hello) UnicastRemoteObject.exportObject(obj,0);
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind("Hello",stub);
    } catch (Exception e) { ... }
  }
}
```

SERVER PROGRAM

Contents

- Introduction
- Remote invocation
- **Message-oriented communication**
- Event-based communication
- Stream-oriented communication

Message-oriented communication

- All communications primitives are defined in terms of passing **messages**

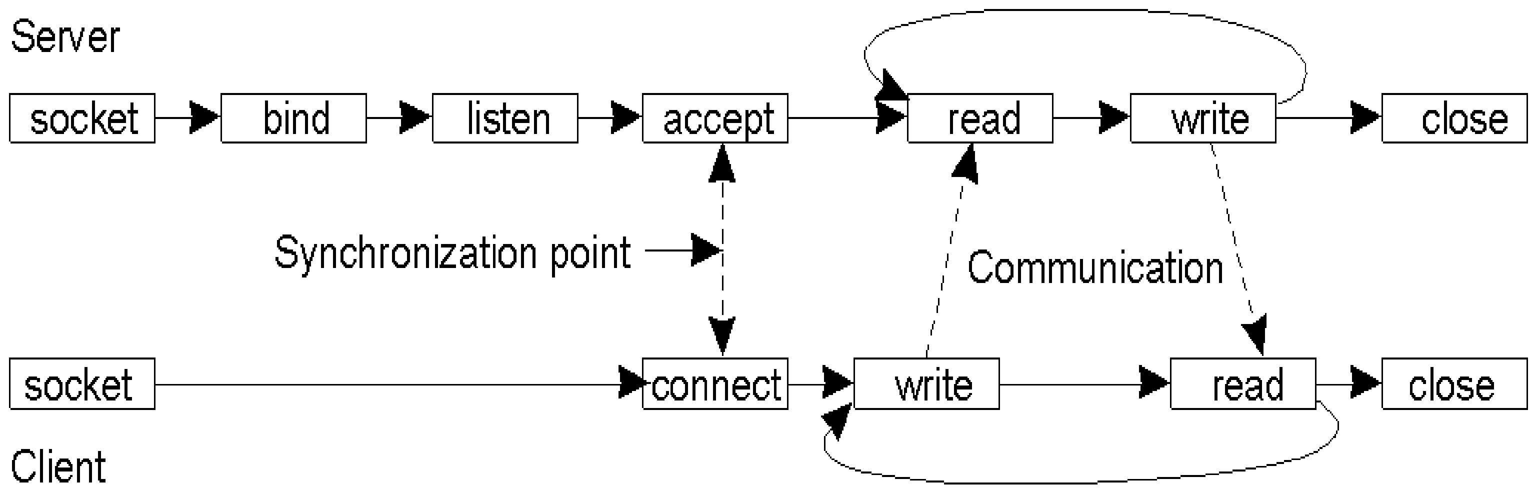
- Transient messaging
  - No support for persistence of messages sent
  - e.g. Sockets
- Persistent messaging
  - Sent messages stored in queue, delivered upon request
  - e.g. Message-Oriented Middleware (MOM)

Berkeley sockets

- API

Primitive	Meaning
socket	Create a new communication endpoint
bind	Attach a local address to a socket
listen	Announce willingness to accept connections
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);
}
```

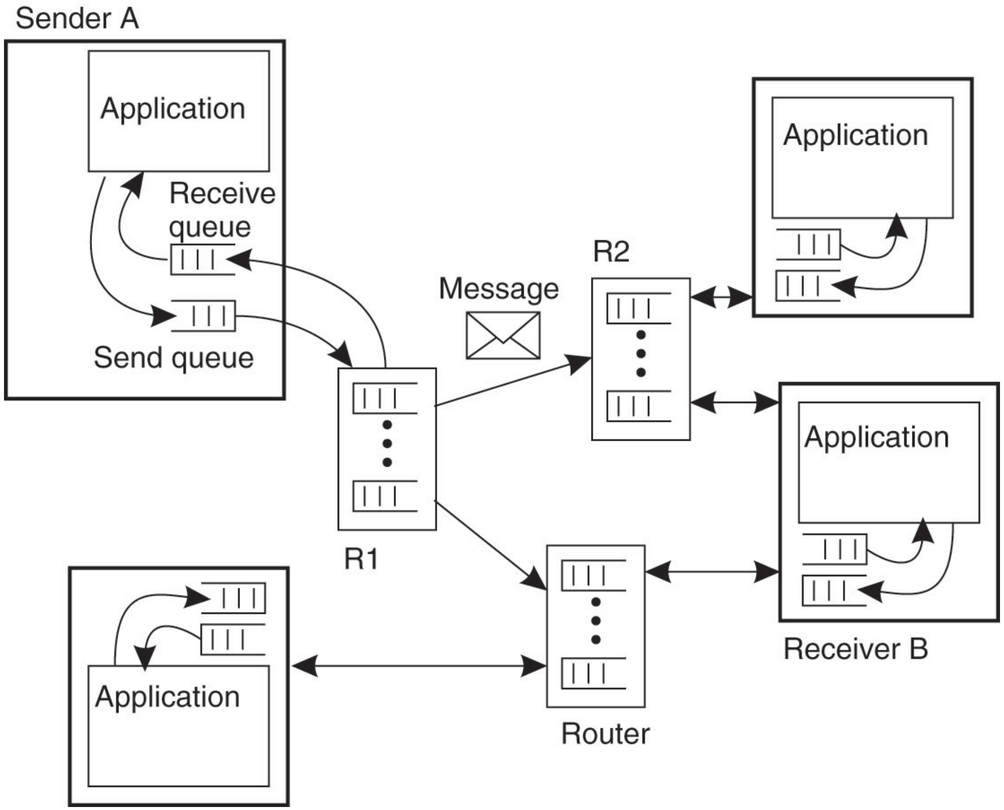
CLIENT PROGRAM

Message-queuing systems

- Supports 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 <u>single</u> process
- a.k.a. Message-Oriented Middleware (MOM)
- e.g. e-mail, Apache ActiveMQ, RabbitMQ

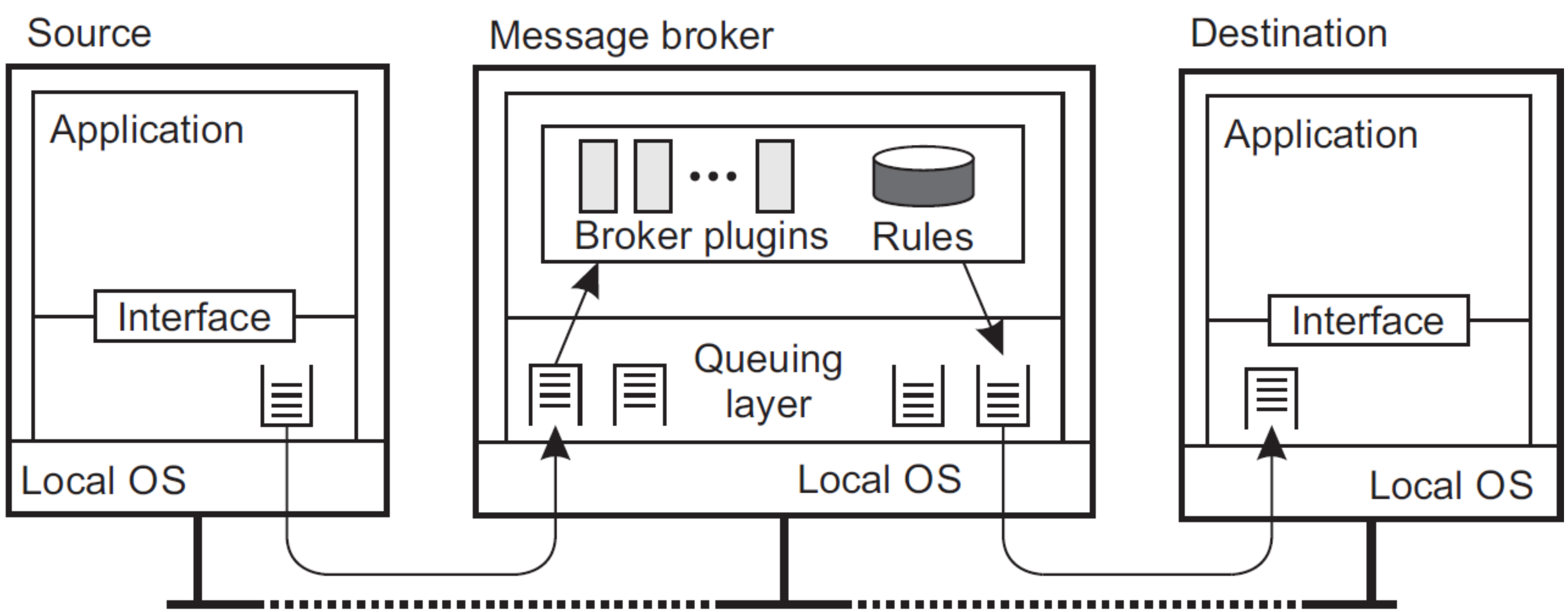
Message-queuing systems

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



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



Message-queuing systems

- API
  - Very simple, yet extremely powerful abstraction

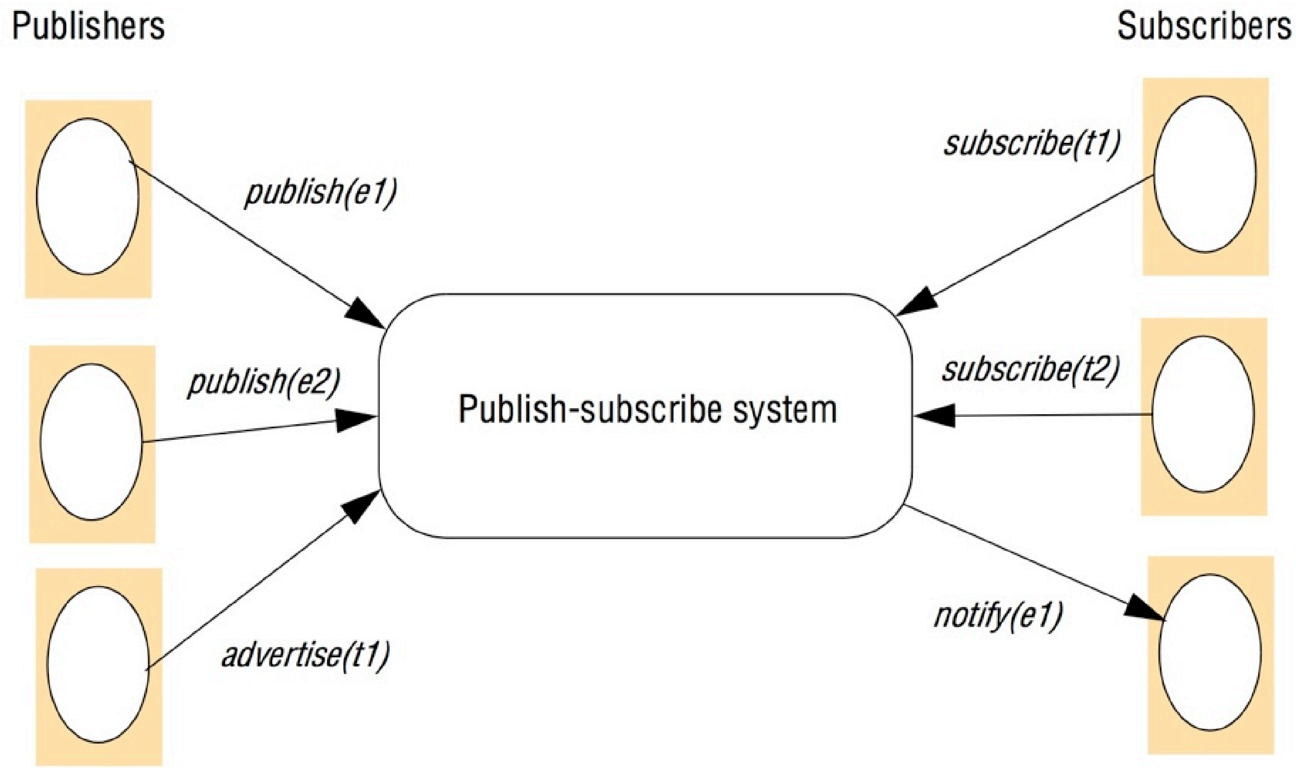
Primitive	**Meaning**
put	Append a message to a specified queue
get	Block until the specified queue is nonempty, and remove the first message
poll	Check a specified queue for messages, and remove the first. Never block
notify	Install a handler to be called when a message is put into the specified queue

Contents

- Introduction
- Remote invocation
- Message-oriented communication
- **Event-based communication**
- Stream-oriented communication

Publish-subscribe systems

- Publishers publish **structured events** to event service
- Subscribers **express interest** in particular events
- Event service matches published events to subscriptions



Publish-subscribe systems

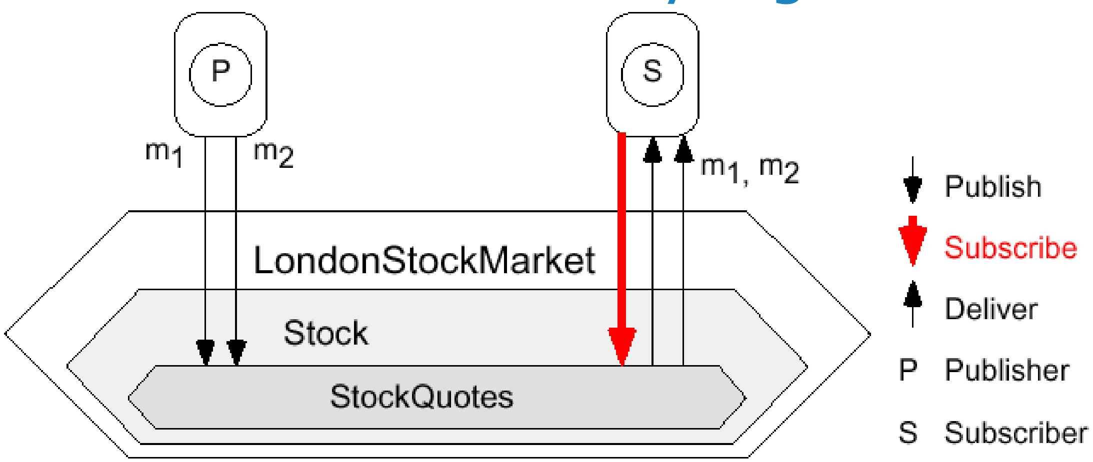
- API

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

Publish-subscribe systems: models

- Subscription (filter) 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



Publish-subscribe systems: models

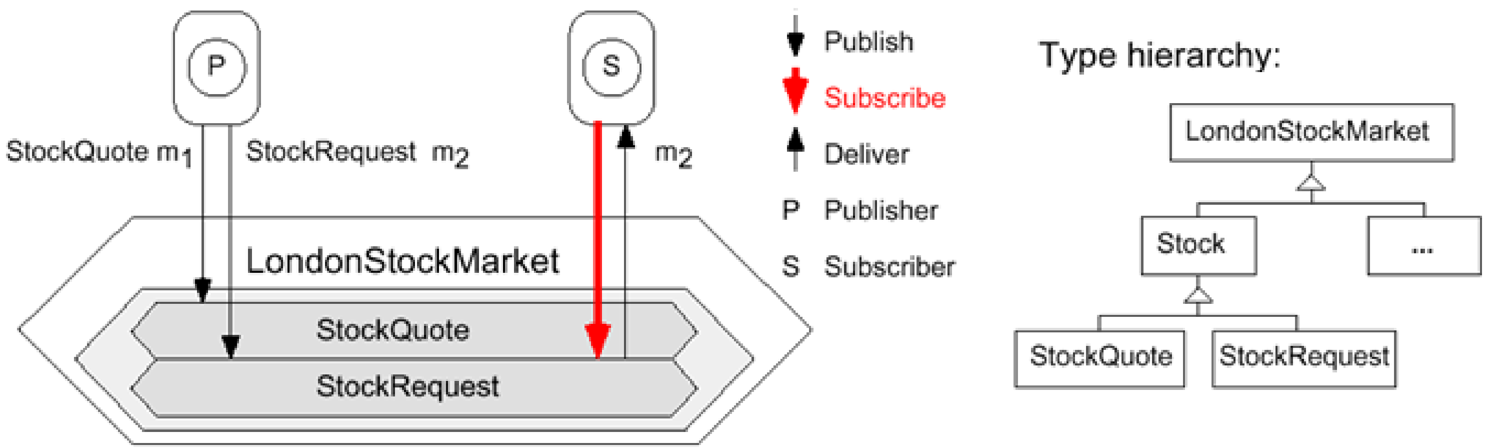
2. Content-based

- Subscribe via compositions of constraints over the values of event attributes
  - Stock quote: (company == ‘TELCO’) and (price < 100)
- More expressive than topic-based

Publish-subscribe systems: models

3. Type-based

- Use object-based approaches with object types
  - Clean integration with OO programming languages
- Subscriptions defined in terms of <u>types of events</u>
- Matching in terms of types or subtypes of the filter



Publish-subscribe systems: architecture

A. Centralized
   – A single node acting as an event broker
   ↑ Simple to implement
   ↓ Lacks resilience and scalability

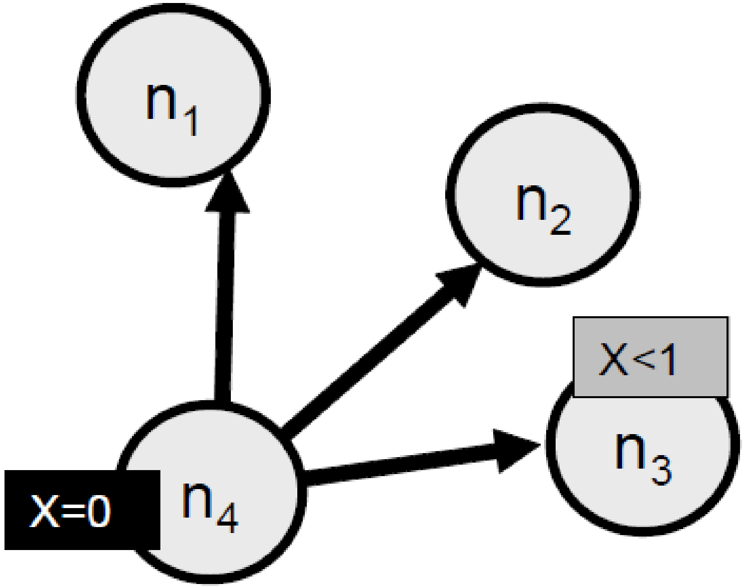
B. Distributed
   a) A network of brokers: each broker stores only a subset of all the subscriptions
   b) P2P: all nodes act as brokers (no distinction between publishers, subscribers, and brokers)
   ↑ Better scalability and fault tolerance

Publish-subscribe systems: routing

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

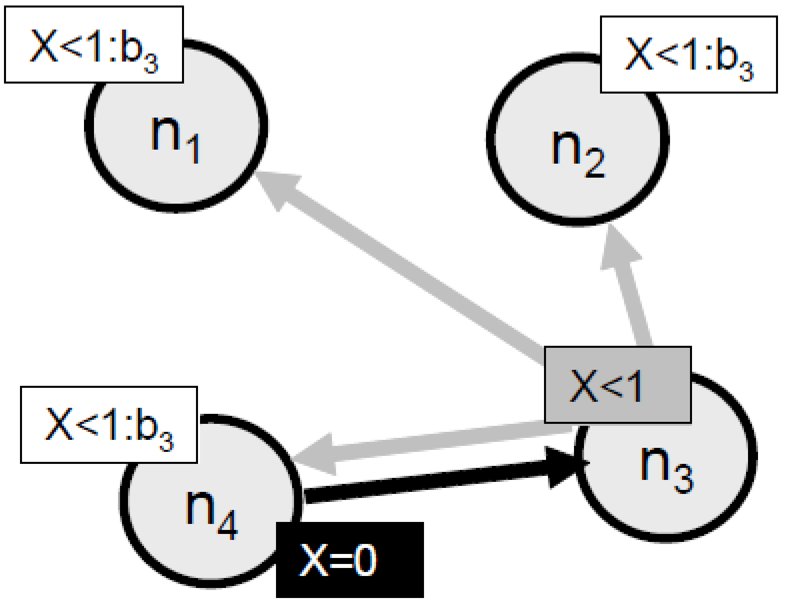
Publish-subscribe systems: routing

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



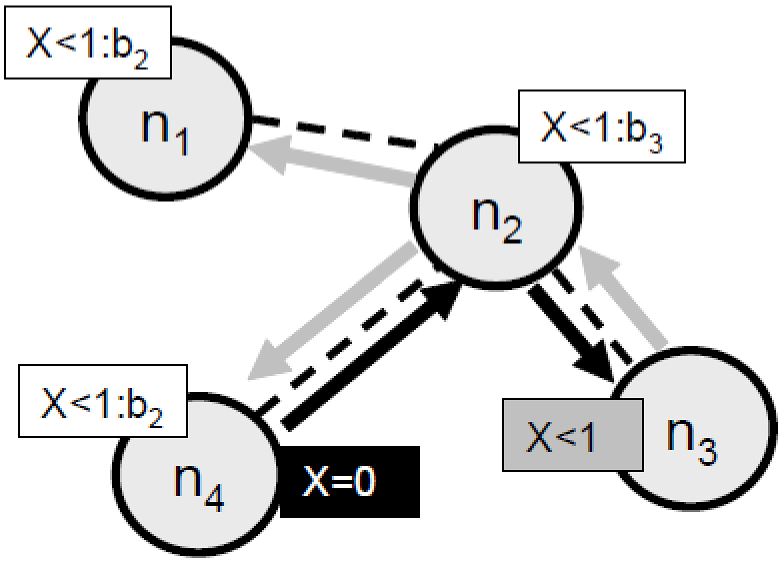
Publish-subscribe systems: routing

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



Publish-subscribe systems: routing

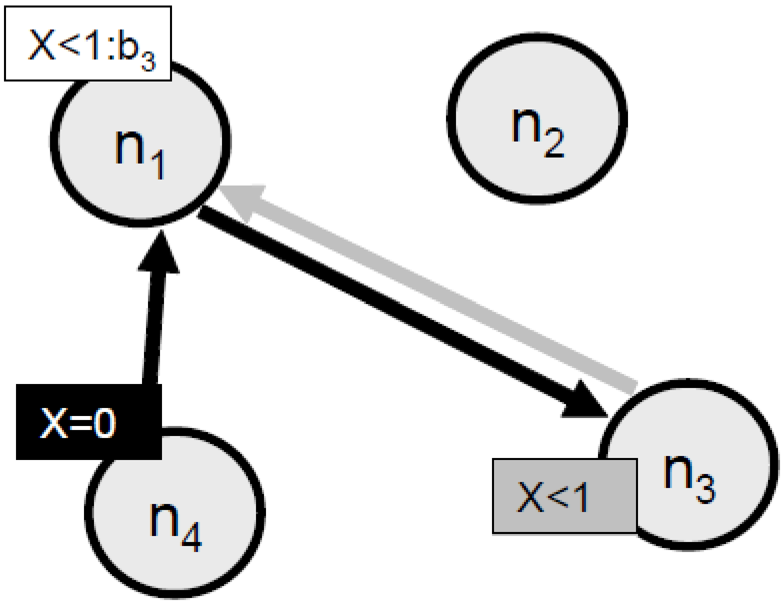
3. Filtering-based
   – Each node stores the set of subscriptions that are reachable through each of its neighbors
   – Send events only to nodes that lay on a path to a valid subscriber
   ↓ Slower event notification
   ↑ Less network traffic and memory use (interaction only with neighbors)



Publish-subscribe systems: routing

4. Rendezvous-based

– Partition subscriptions & events among nodes
  - SN(σ): nodes storing σ
  - EN(e): nodes matching e against subscriptions
  - $e \in \sigma \rightarrow$ SN(σ) ∩ EN(e)≠Ø
  - e.g. use a distributed hash table (DHT)



– Send event to nodes ∈ EN

↓ Rearrange subscriptions when nodes join/leave

↑ Balanced subscription storage and management

Publish-subscribe systems: 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
   ↓ Redundancy in message traffic
   ↑ Simple, no memory overhead
   ↑ Supports highly dynamic systems
   ↑ Scales well

Contents

- Introduction
- Remote invocation
- Message-oriented communication
- Event-based communication
- **Stream-oriented communication**

Stream-oriented communication

- With RPC, RMI and MOM timing has no effect on correctness
  - **Time-independent** communications
- **Time-dependent** communications
  - Timing is crucial. If wrong, the resulting 'output' from the system will be incorrect
  - e.g. audio, video, animation, sensor data
  - a.k.a. 'continuous media' communications
  - Examples:
    - audio: PCM: 1/44100 sec intervals on playback
    - video: 30 frames per second (30-40 msec per image)

Stream-oriented communication

- Transmission modes
  - Asynchronous transmission mode
    - Data units are transmitted in order with an arbitrary delay between them (e.g. file transfer)
  - Synchronous transmission mode
    - There is a maximum end-to-end delay for each data unit, but data can travel faster (e.g. real-time sensor data)
  - Isochronous transmission mode
    - There is a **maximum** and a **minimum** end-to-end delay (a.k.a. 'bounded jitter') for each data unit
      - Data units must be transferred 'on time'
      - e.g. multimedia systems

Stream-oriented communication

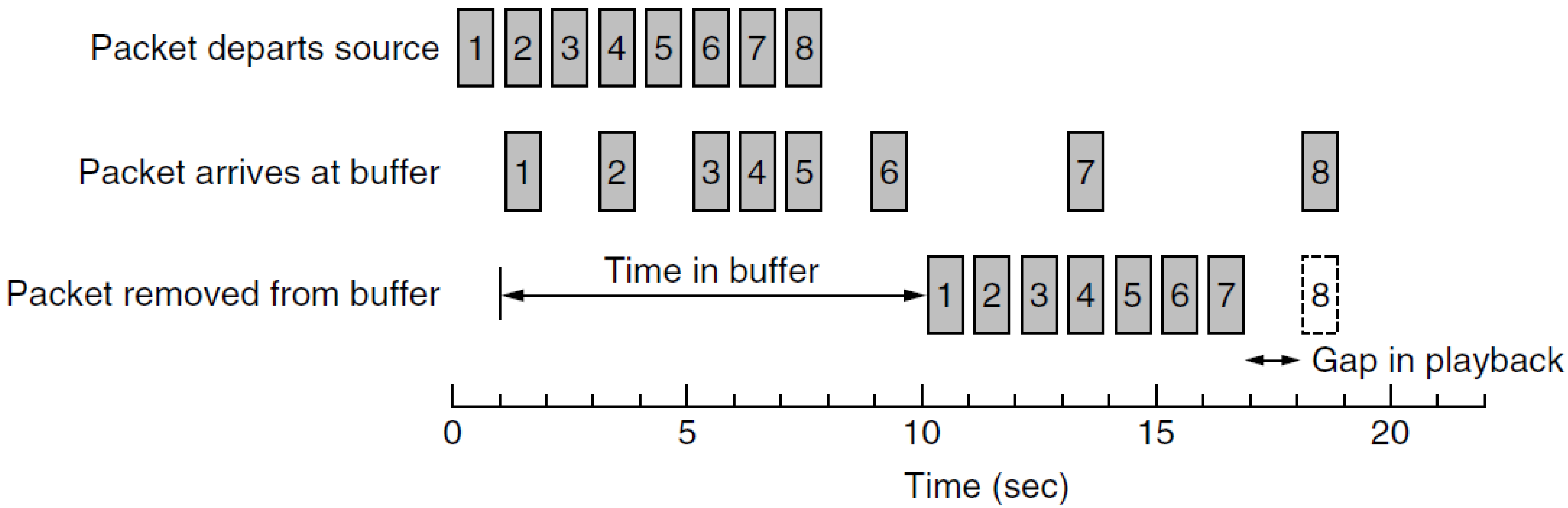
- Stream
  - An unidirectional continuous data stream that supports isochronous data transmission
  - Generally a single *source*, and one or more *sinks*

A. Simple streams
   - One single sequence of data (e.g. audio or video)

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

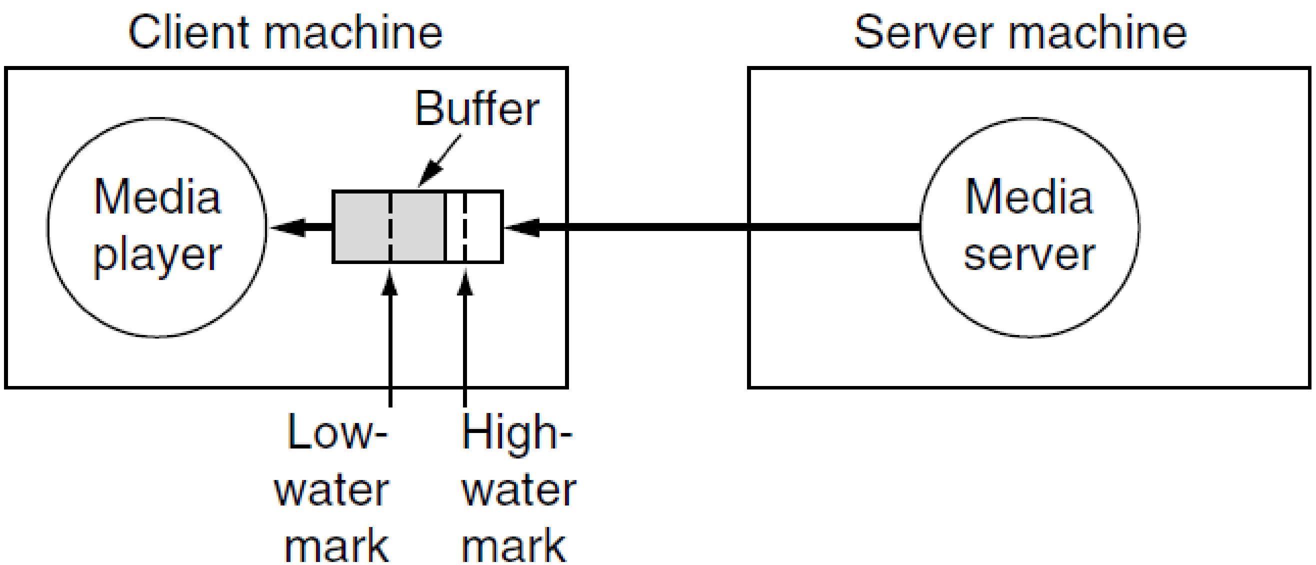
Enforcing timing (QoS)

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



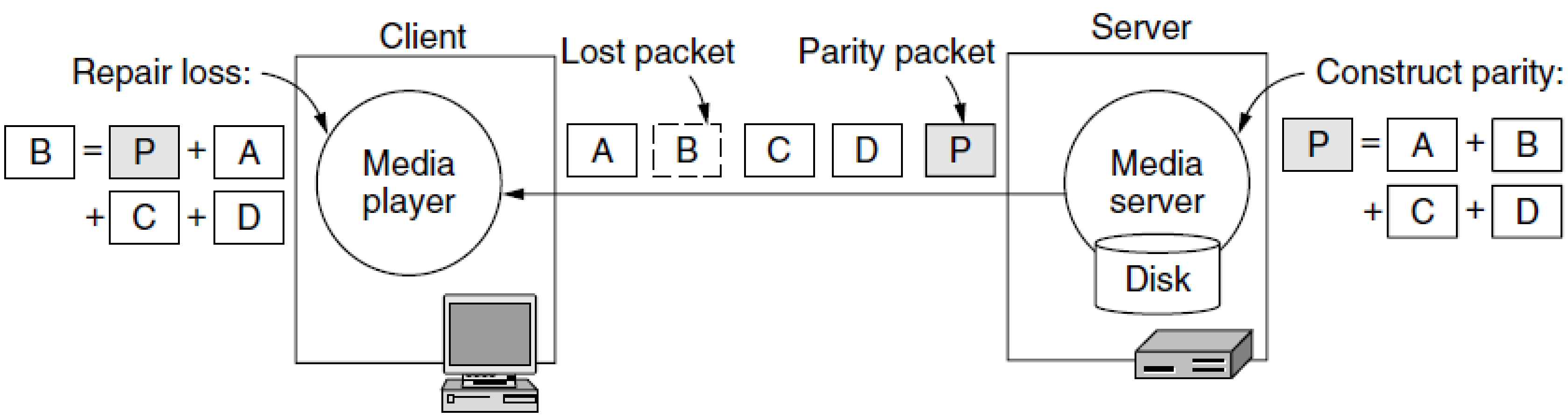
Enforcing timing (QoS)

1. Buffering
   - Low-water mark: Determines the minimum amount of data in the buffer to start to play
   - High-water mark: Defines when the client will ask the server to pause the transmission



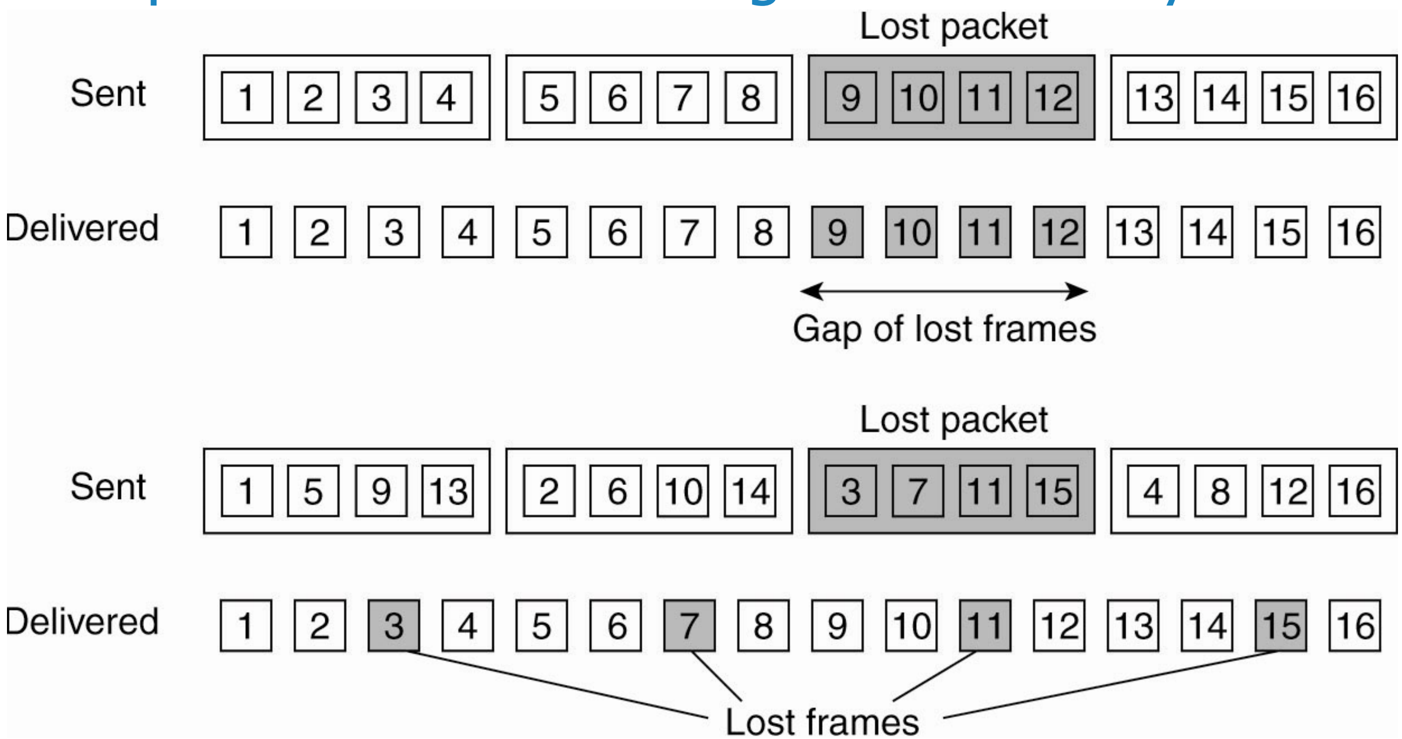
Enforcing timing (QoS)

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



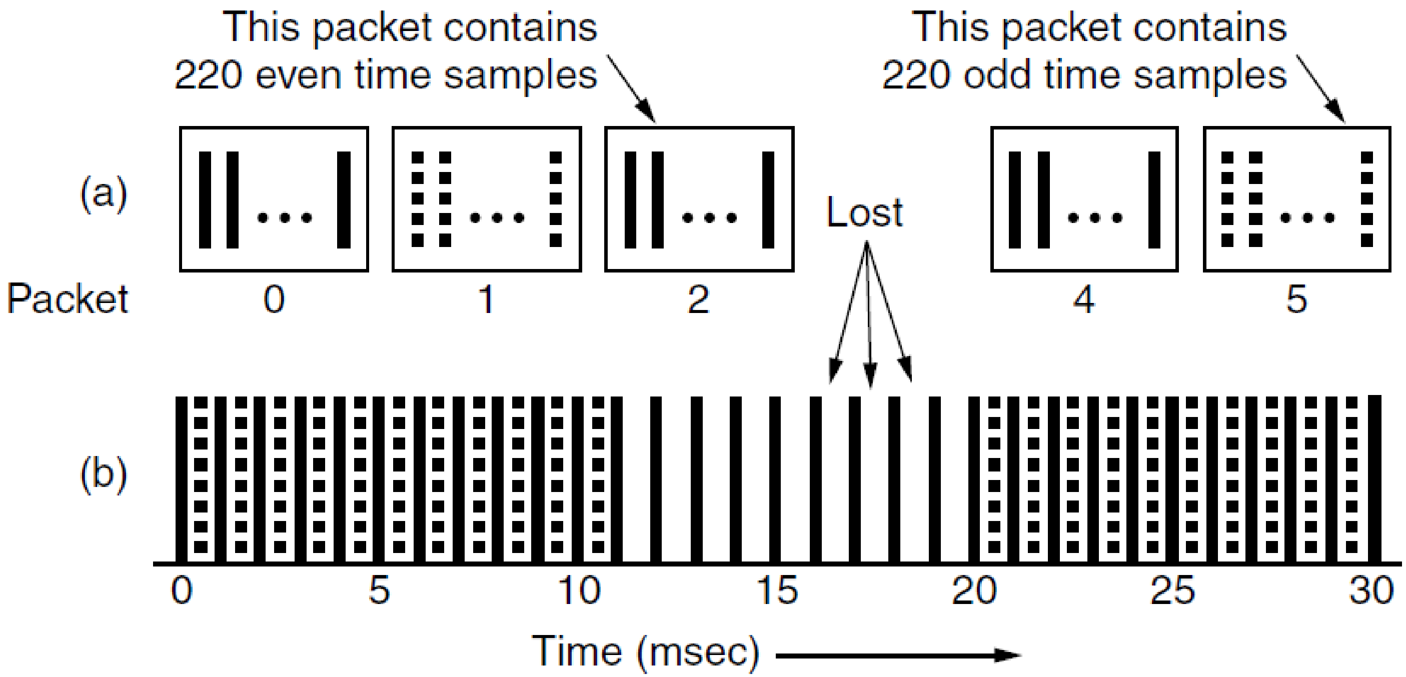
Enforcing timing (QoS)

3. Interleaved transmission
   – Reduce the impact of packet losses
   – Gap is distributed but 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 and RMI: synchronous, transient
- MOM: asynchronous, persistent
- Publish-subscribe: decoupled in space
- Streams: for ‘temporally-related data’
- Further details:
  – [Tanenbaum]: chapters 4, 8.3, and 10.3
  – [Coulouris]: chapters 4, 5, 6, and 20