Distributed Systems

6 RMI/MP IPC

May-18-2009

Gerd Liefländer

System Architecture Group

Intended Schedule of Today

- RMI (only rough overview)
- Message Passing
  - Motivation
  - Bridge Principle
- Message Passing Systems
  - Design parameters
  - Direct versus indirect naming
  - Transient versus persistent communication
  - Persistent communication
  - IPC Semantics
  - Communication Endpoints
- Events & Notification (see additional slides)
- Stream-oriented Communication (see other courses)

Remote Method Invocation

RPC → RMI

RPC: invoke procedure of a remote server

RMI: invoke method of a remote object at a remote server (quite similar to RPC)

Remote Method Invocation (RMI)

- Client side stub: proxy marshalling method parameters and transferring message to the server machine
- Server side stub: skeleton unmarshalling parameters and calling remote object and "back again"
- Piece of software at "server side" is a remote object (if its state is completely located at one remote machine)
- Sometimes even the state of a remote object can be distributed, then this is a true distributed object
- Additional problems arise when objects can migrate and/or are replicated

RPC versus RMI

<u>Remote object</u>:

- Not associated to a specific server node, i.e. we can achieve better transparency if object can migrate

- Remotely accessible
  - Exchange remote references between tasks

- Encapsulates state
  - Easier to keep track of related state
  - Easier migration/replication of state
  - Easier synchronization of concurrent RMIs

CORBA IDL Example

```
// In file Person.idl
struct Person {
   string name;
   string place;
   short salary;
   long year;
} ;

interface PersonList {
   readonly attribute string listname;
   void addPerson(in Person p) ;
   void getPerson(in string name, out Person p);
   void updatePerson(in Person p, short s, out Person p);
   long number();
};
```

Message Passing

Major drawbacks of RPC (and RMI):

1. Only transient communication, i.e. if callee is not activated request is lost
2. Caller synchronously blocked
3. Application programmers sometimes want more flexibility

Why do we need Message Passing?

- RPC/RMI is too restricted (e.g. limited parameters) or just not available on all involved nodes
- Sometimes applications need a comfortable and flexible IPC mechanism, e.g. a multicast to notify multiple nodes, e.g. in order to speed up a search
- Applications want to cooperate asynchronously
  - Consumer on $node_1$, producer on $node_2$
  - P2P applications
  - Client-Server applications

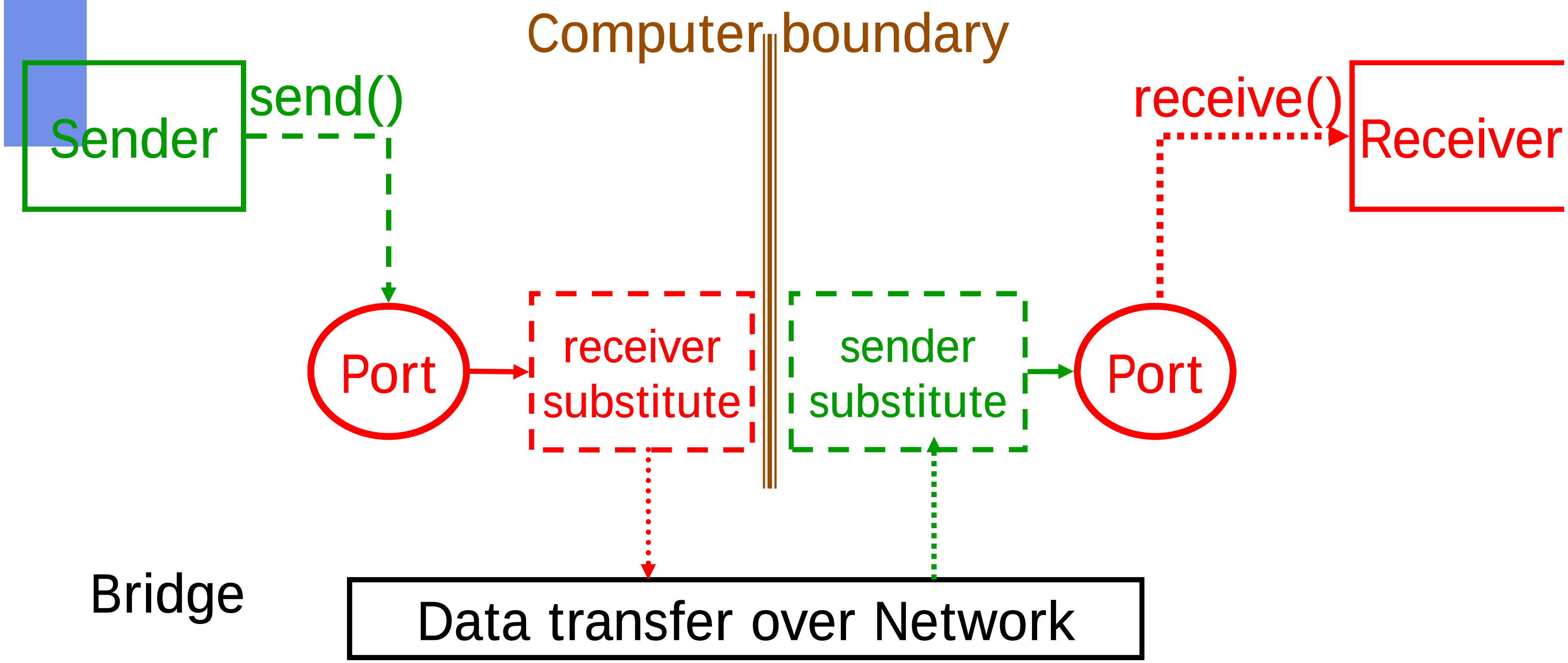
⇒ We need an efficient & secure way to transfer application messages across the network(s)

Typical IPC Applications

- IPC via message passing is appropriate for fast cooperation between distributed applications, i.e.
  - Synchronize physical clocks
  - Enable mutual exclusion
  - Elect a new coordinator
  - Enable a global snapshot
  - Detect a global deadlock
  - ...
- In all these cases a synchronous RPC is insufficient

Bridge Principle

Communication Bridge Principle



Remark: The functionality of the substitutes may vary depending on the requirements of the “pair” <sender(s)/receiver(s)>

How to implement Communication Bridges?

- Bridges completely outside of the kernel

⇒ distributed process model (Prozessverbund[1])

- Bridges within kernel & outside of kernel

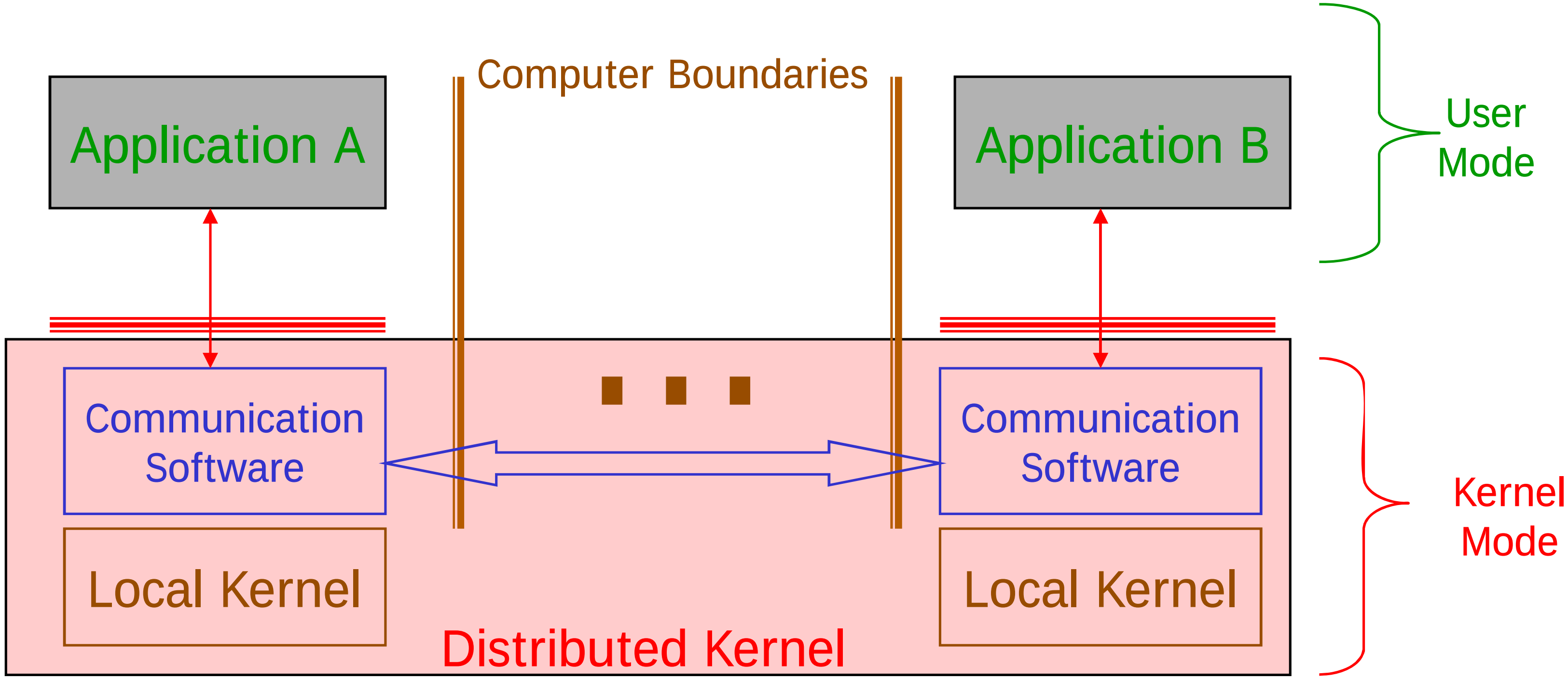
⇒ Hybrid model

- Bridges completely within kernel

⇒ distributed kernel model (Kernverbund)

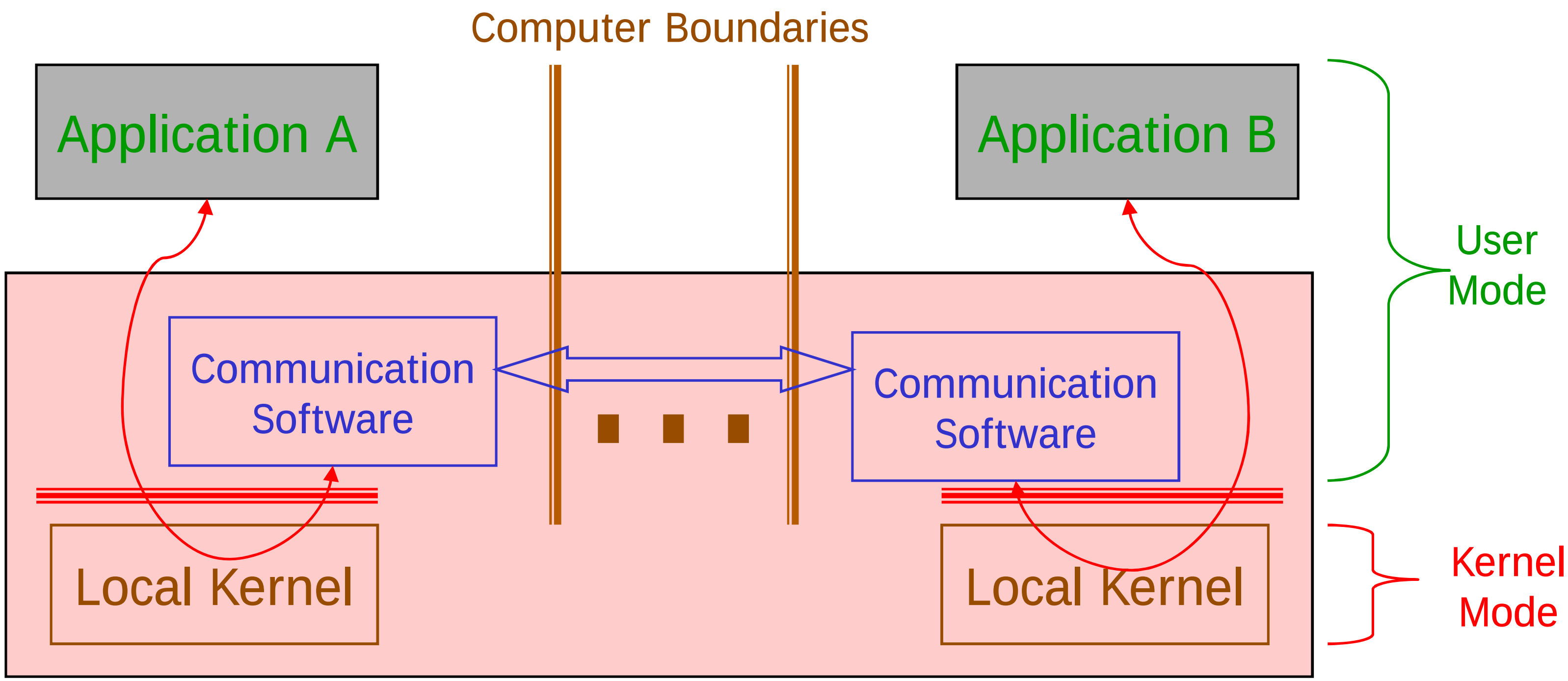
[1]Terminology of Heiss (TU Berlin) and Wettstein

Distributed Kernel Model



- Distribution is hidden below the kernel API
- System calls at kernel API might access arbitrary kernel-objects
- Distributed kernel := union of all local kernels

Distributed Process Model



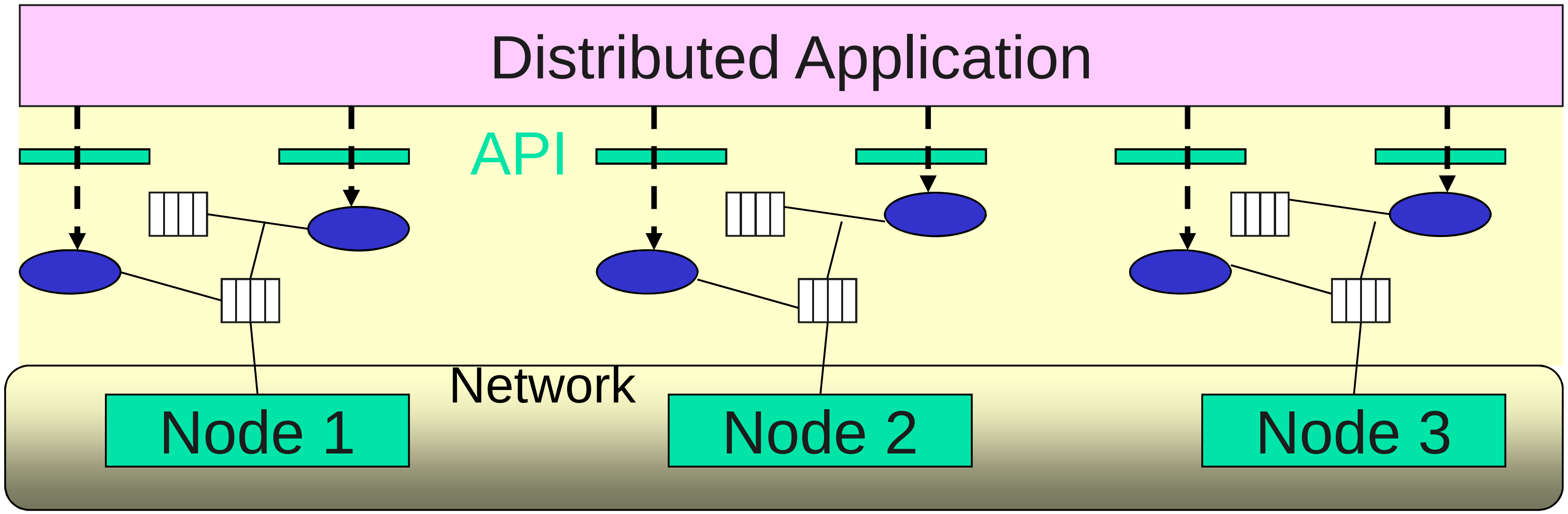
- API of local kernels must not be adapted
- Local kernel is not aware of being a member of a DS

IPC Models & Parameters

- Introduction
- Simple or Multiple Client Server
- Message-Oriented Transient IPC
- Sockets and MPI
- Message-Oriented Persistent IPC

Message Passing System*

- Implements explicit data transfer via a network
- Offers communication primitives at API at least
  - a `send(...)` & a `receive(...)` operation

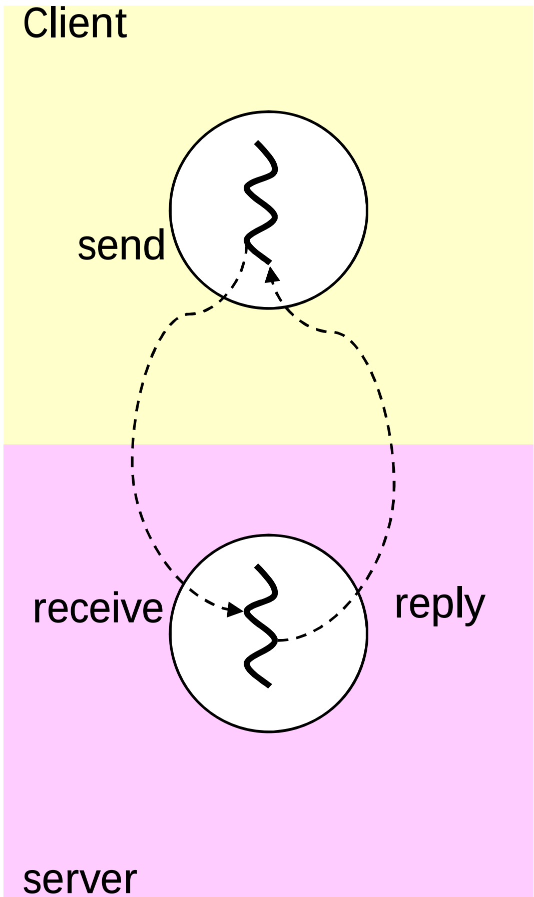


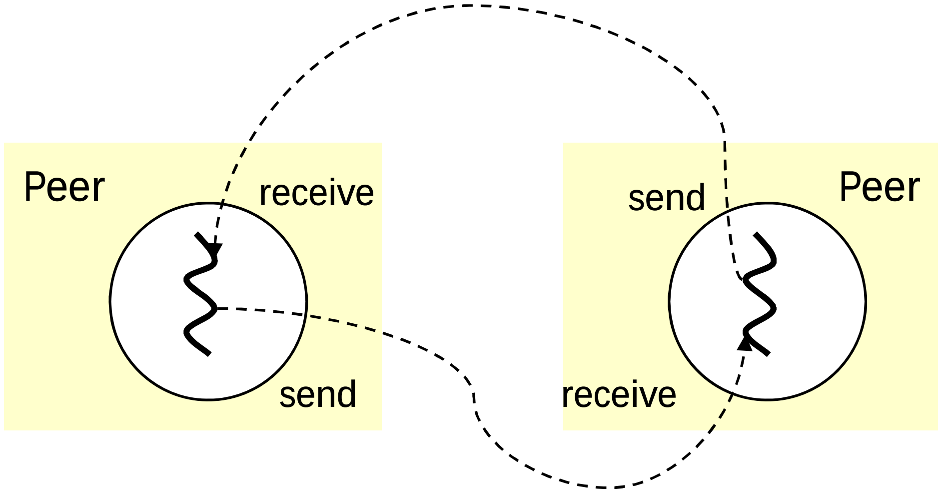
*Very simple form of a middleware

Simple IPC Models

- Only 2 instances are part of the IPC, e.g.
  - 1 client and 1 server, e.g.
    - 2 processes or
    - 1 process and a procedure
  - Clients requests, server replies
  - 2 peers (always 2 processes), both partners with equal communication rights can send or receive

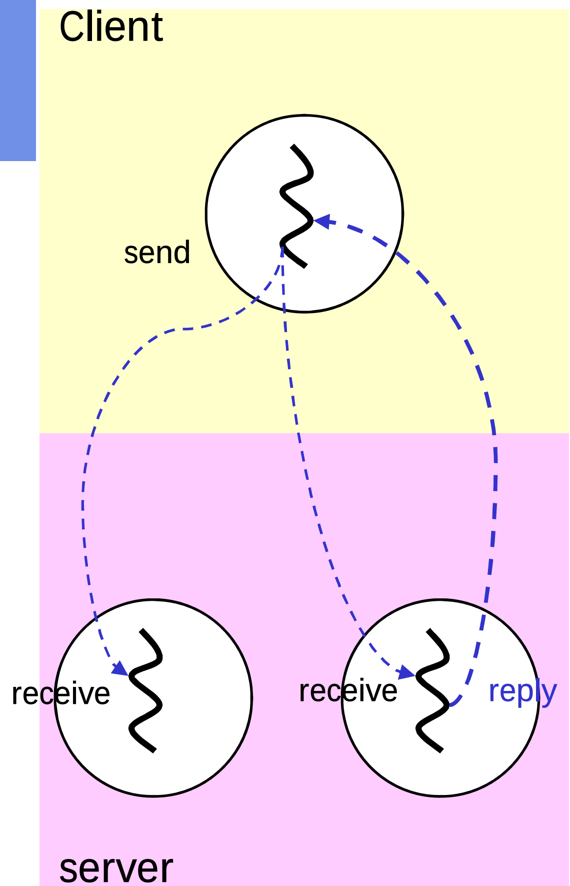
Simple IPC Pattern



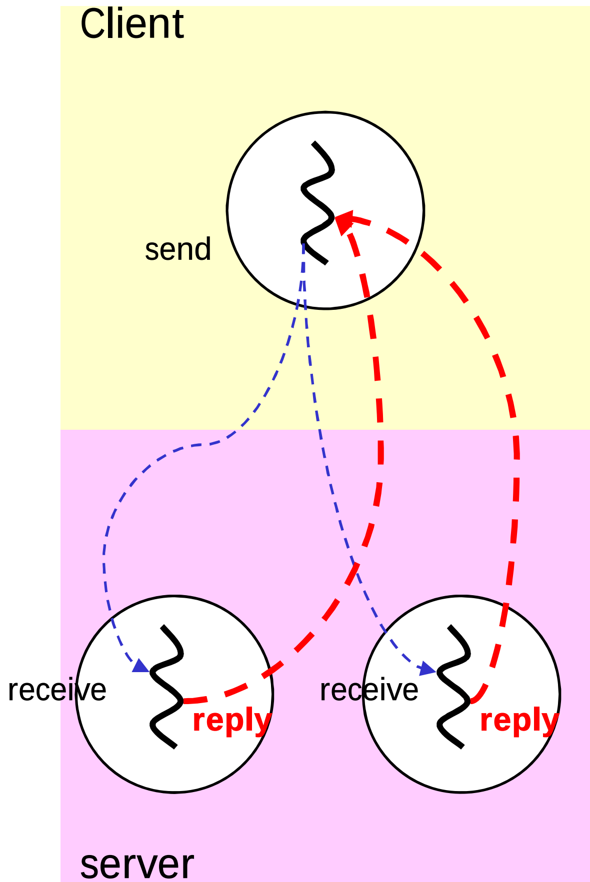


sendRequest, receiveRequest, replyResult

Multiple IPC Pattern (Client/Server)



sendRequest as multicast



What to do when replies are different?

Pragmatic Design Parameters

- Length of message
  - Constant or fixed
  - Variable, but limited in size
  - Unlimited
- Loss of messages
  - Not noticed
  - Suspected and notified
  - Avoided
- Integrity of messages
  - Not noticed
  - Detected and notified
  - Automatically corrected

Orthogonal Design Parameters

- Number of involved communication partners
- Synchronous versus asynchronous
- Placing the message buffers
- Persistent versus transient communication
- Addressing the communicating instances
- ...

Direct versus Indirect Addressing

Relationship mailboxes (ports) & processes

- 1:1 (one port per process)
- 1:n
- m:1
- m:n

- Extension of buffering (in mailbox, channels)
  - Number of involved buffers
  - Limited buffer size (typical)

Direct Addressing

- Source and destination process/task serve as designators
  - Allows a process *easy control* when to receive a message from a specific process
  - Used to implement client/server applications
    - Well suited iff 1 client & 1 server
    - Otherwise: server must be able to receive request from any client
    - A client should be allowed to invoke many services at a time if more than one server is available

Indirect Addressing (Mailboxes)

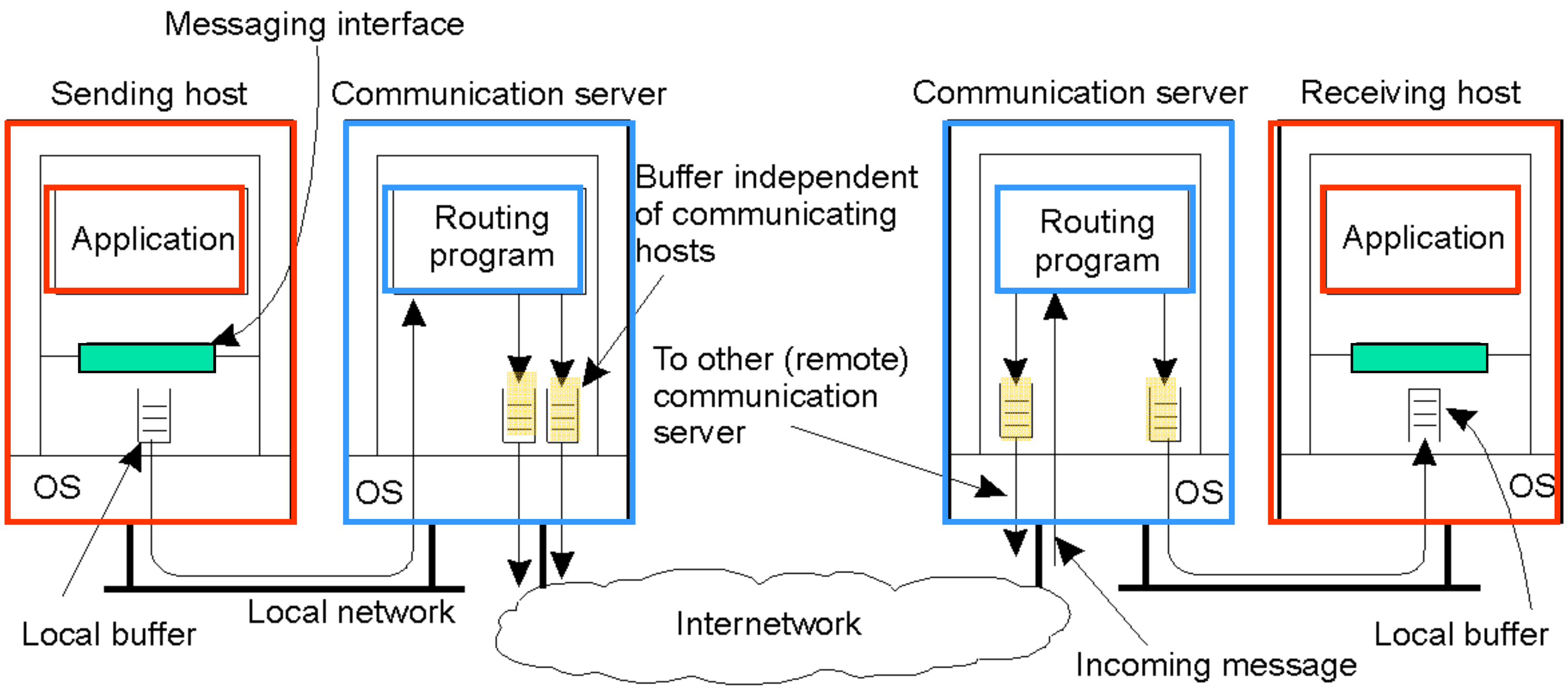
- Mailbox shared by n>1 processes
  - Messages sent to a mailbox can be received by any process currently attached to mailbox
  - Implement more flexible client/server applications
    - Client sends request to mailbox next server executes it
  - Drawback: costly implementation
    1. Message sent to mailbox
    2. Relayed to all other sites that could potentially receive from mailbox
    3. If on site decides to receive, inform all other sites that message is no longer available for receipt
    4. Mutual exclusion for concurrent access

Indirect Addressing (Cont.)

- Port=mailbox, only one process can receive from
  - ....
  - ...

- Channel
  - Static (at compile time)
    - ...
  - Dynamic (at runtime)
    - ...

Persistence & Synchronicity

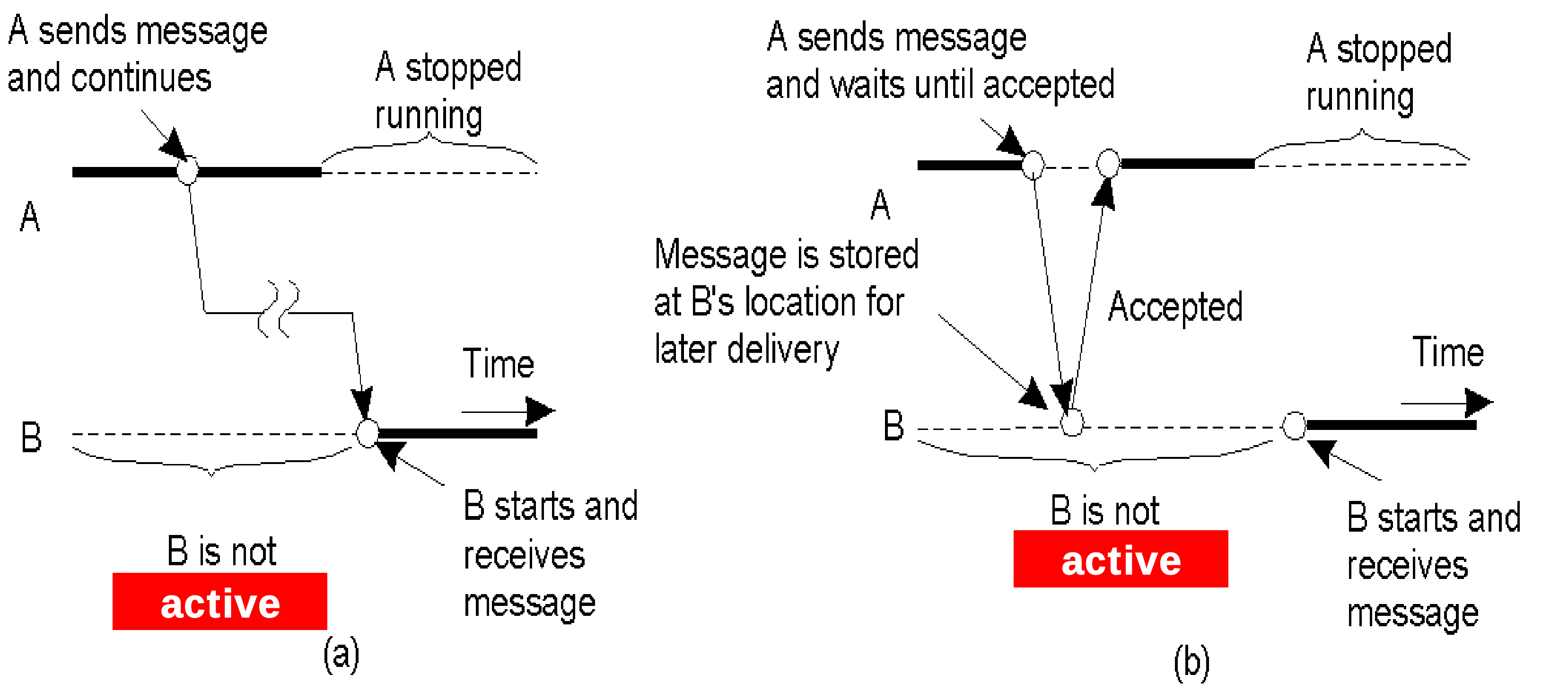


- Organization of a communication system in which hosts are connected through communication servers via a network
- Communication servers (and/or hosts) can hold undeliverable messages as long as needed

Communication Models

- Persistent versus Transient
  - Persistent messages stored as long as necessary by the communication system (e.g. E-mail)
  - Transient messages are discarded when they cannot be delivered (e.g. transport level)
- Synchronous versus Asynchronous
  - Asynchronous implies sender proceeds as soon as it sends the message, i.e. no blocking
  - Synchronous implies sender blocks until the receiver buffers the message or even delivers the message to the receiver

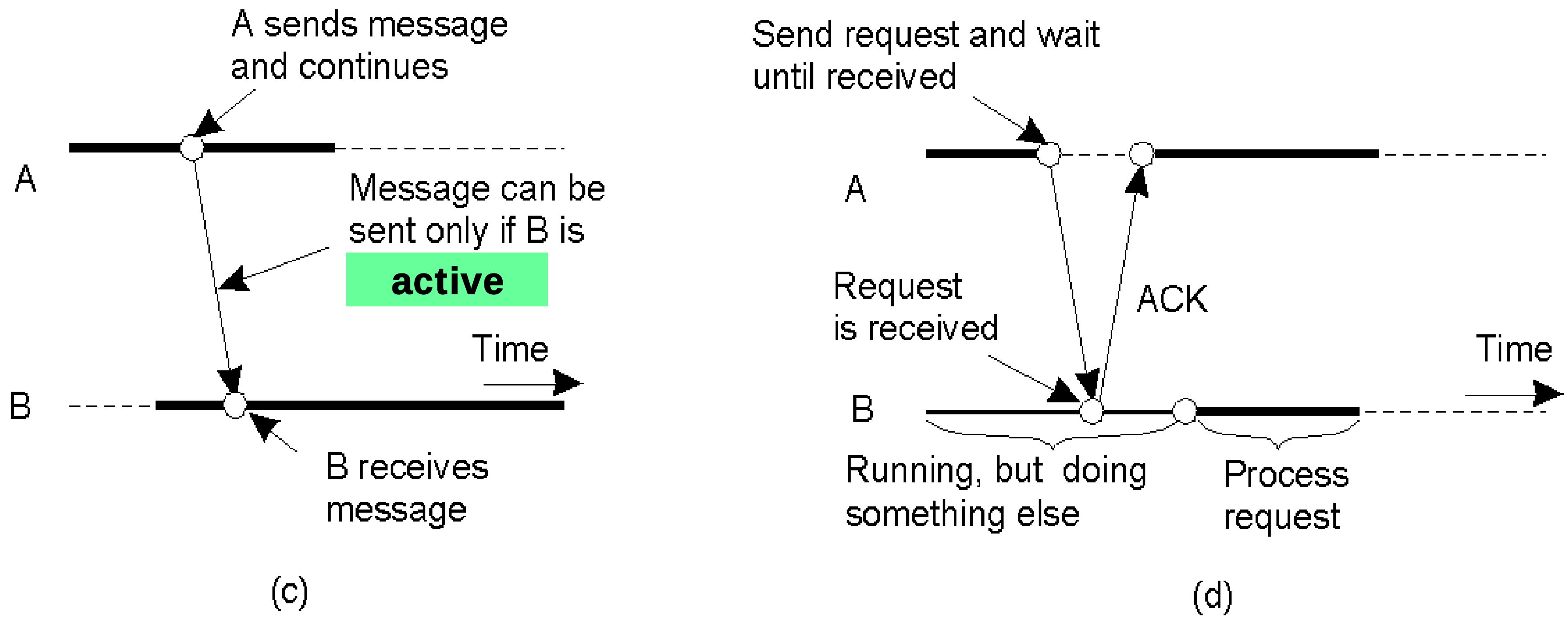
Persistence and Synchronicity (1)



a) Persistent asynchronous communication (email)

b) Persistent synchronous communication

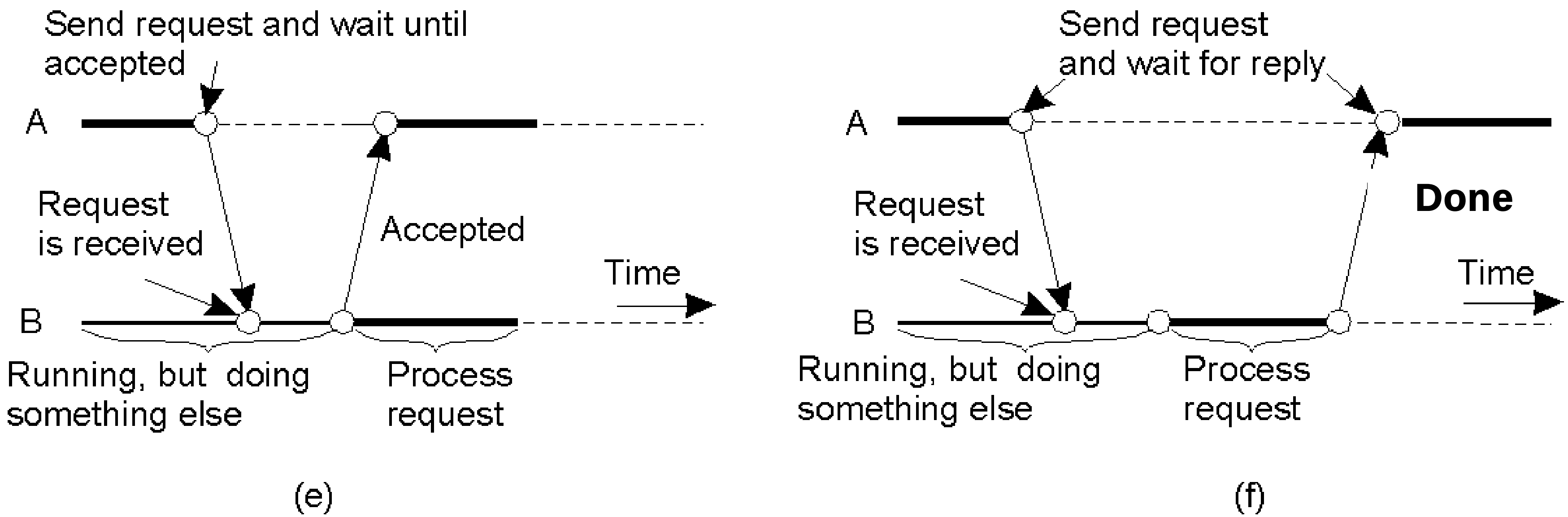
Persistence and Synchronicity (2)



c) Transient asynchronous communication

d) Receipt-based transient synchronous communication

Persistence and Synchronicity (3)



e) Delivery-based transient synchronous communication

f) Response-based transient synchronous communication

Transient IPC

Berkeley Sockets

MPI

Message Passing Interface (1)

- Overcome disadvantages of sockets:
  - Wrong level of abstraction being implemented at a too low level with only very primitive operations
  - Designed for communication across networks using general-purpose protocol stacks (TCP/IP)
  - Relatively poor performance

⇒ Not well suited for high-speed interconnection networks used in COW* (Myrinet)

*COW = Cluster Of Workstations

MPI (2)

Assumptions:

1. Communication only within a group of processes
2. Each group has a unique identifier
3. Groups may overlap
4. Each process in a group has a (local) identifier

⇒ <GID, PID> identifies source/target of message

5. Support diverse forms of buffering and synchronization (over 100 functions)
6. If serious failures occur (e.g. network partition), no automatic recovery is offered

Message-Passing Interface (3)

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

- The most intuitive message-passing primitives of MPI

Persistent IPC

Message-Queuing Systems

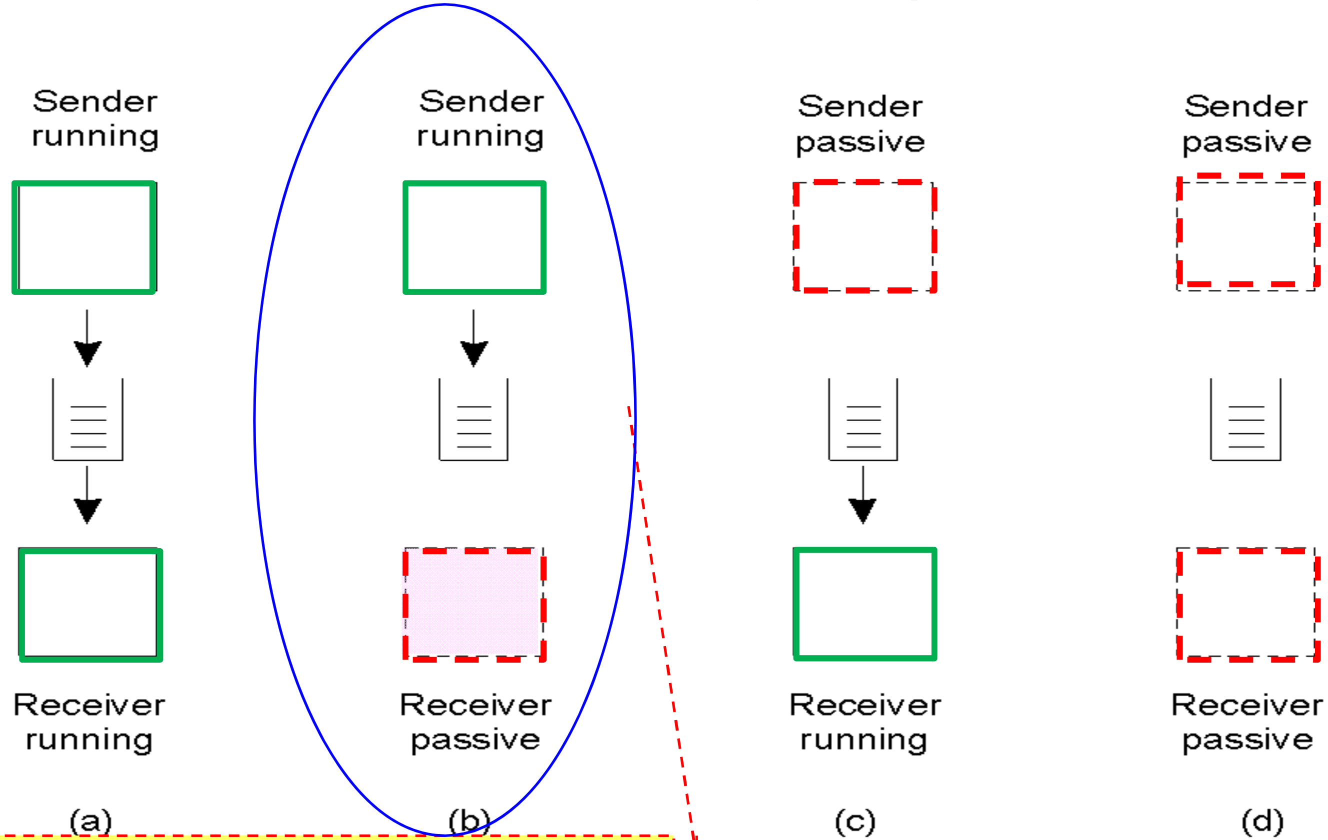
Message-Oriented Middleware (MOM)

Persistent Communication

- Application communicate by inserting messages in specific message queues
  - Loosely coupled communication, i.e. it’s no longer required that both sides are active while communicating
  - Offer persistent intermediate-term storage capacity
- Applications can usually tolerate longer message transfer times
- Applications typically need larger message sizes

Message-Queuing Model (1)

- 4 combinations for a loosely-coupled communication



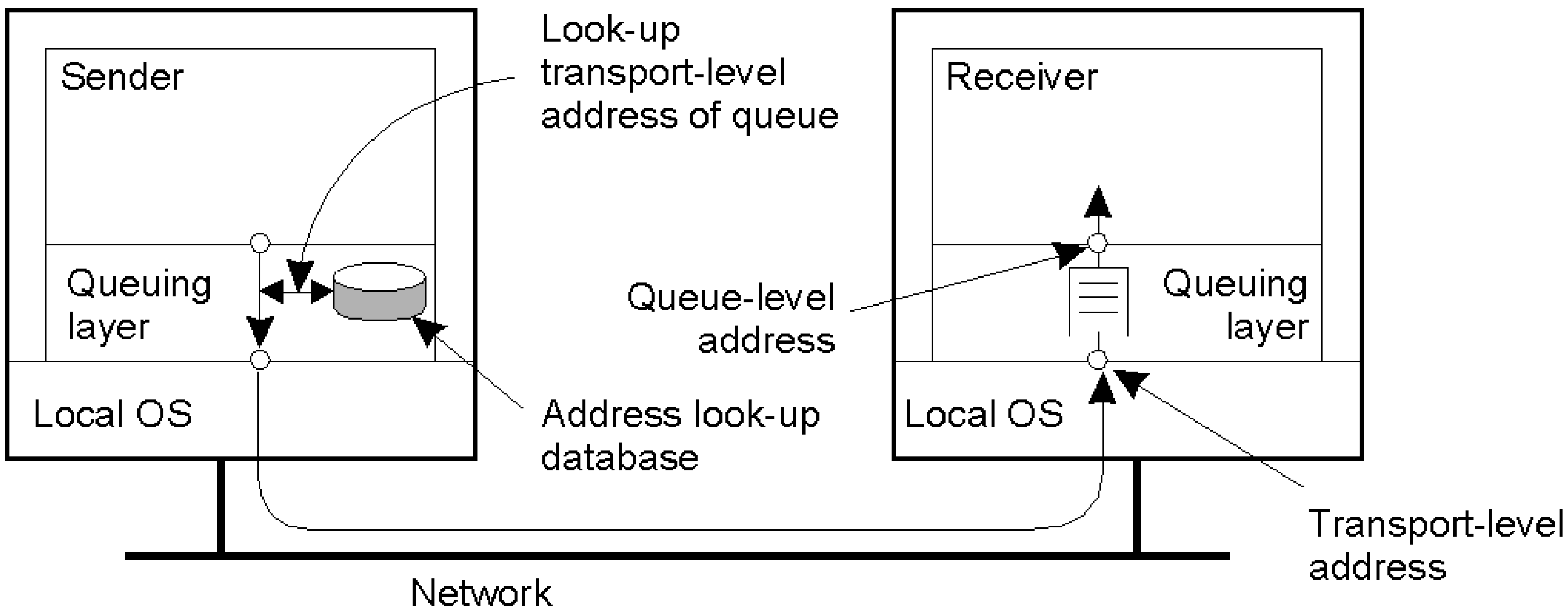
(b) Message remains in one of the queues till receiver is calling a receive, sender keeps on sending

Message-Queuing Model (2)

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 an observer handler to be called when a message is put into the specified queue (callback function).

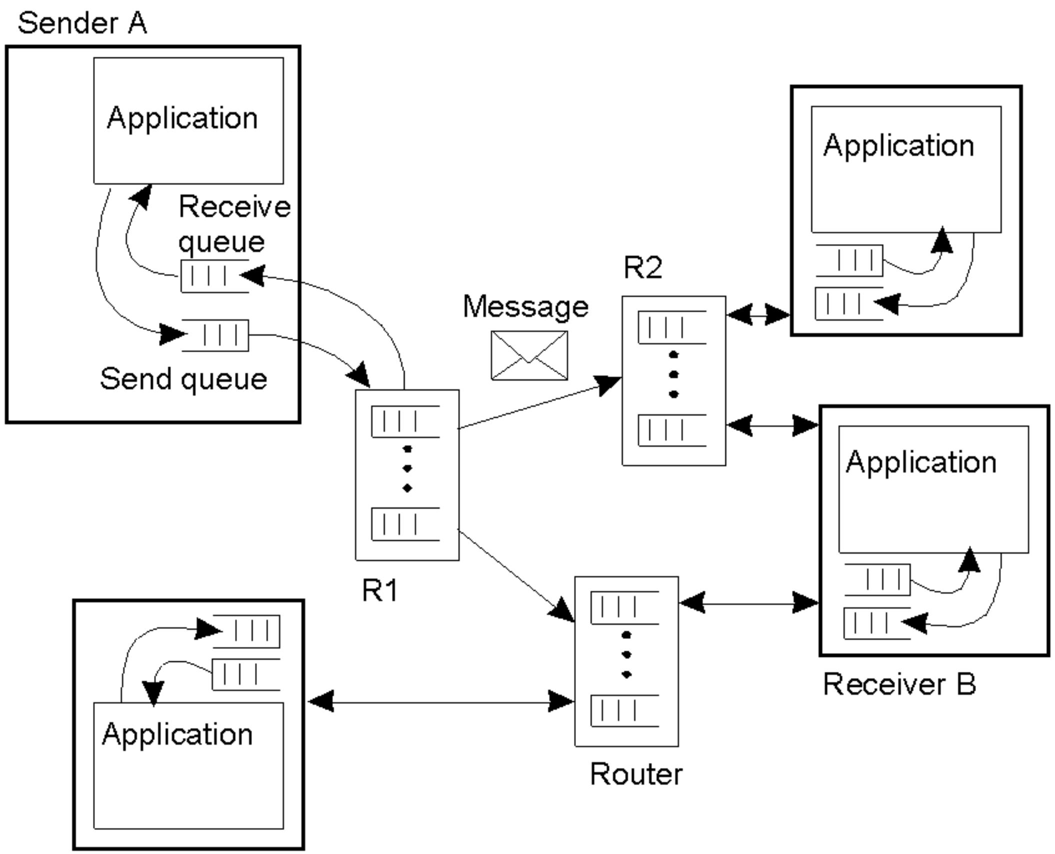
- Basic interface to a queue in a message-queuing system.

Message-Queuing System (1)



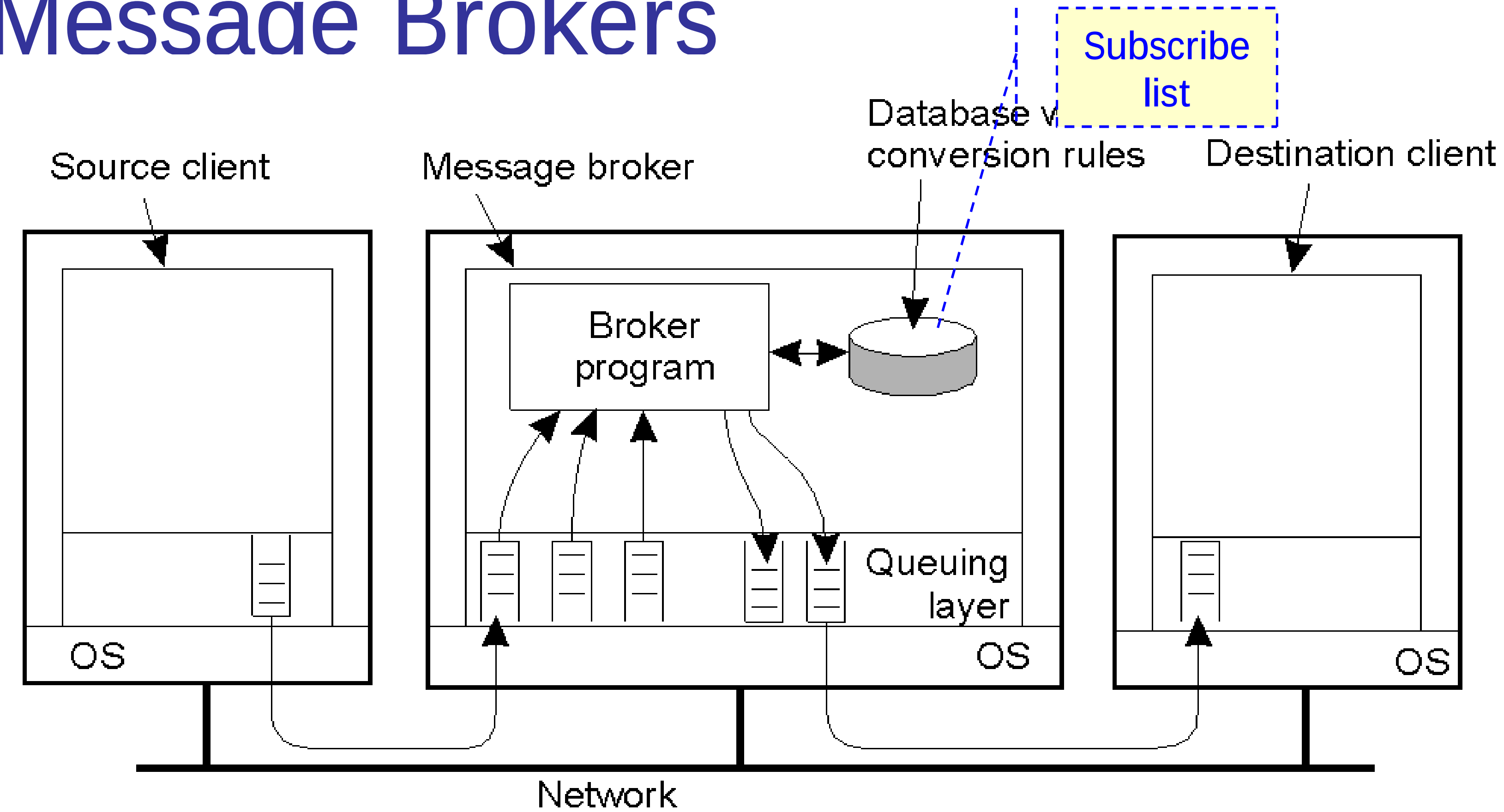
- Relationship between queue-level addressing and network-level addressing

Message-Queuing System



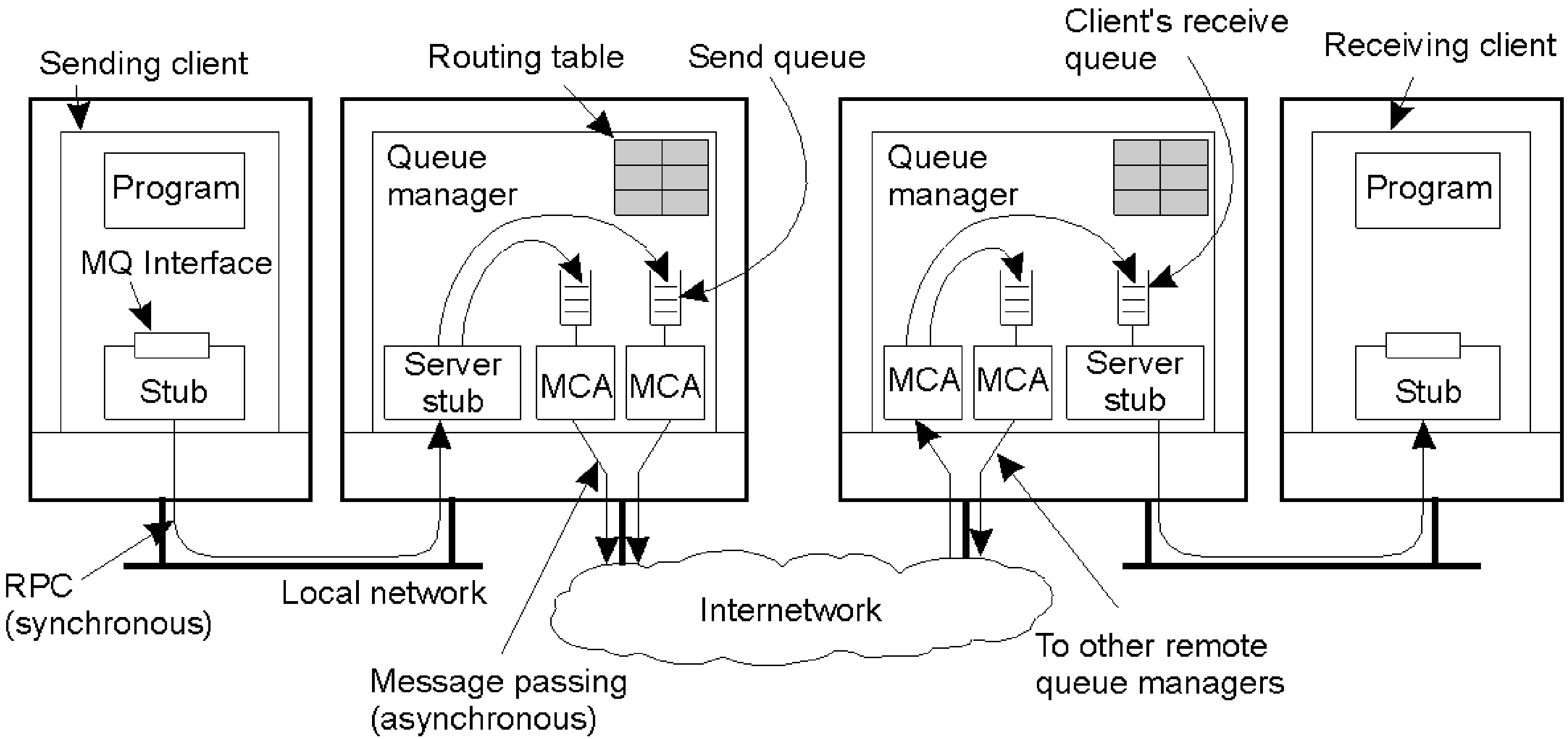
- General organization of a message-queuing system with routers

Message Brokers



- General organization of a message broker in a MQS
- A message broker can also act as a central manager of a publish/subscribe systems

Example: IBM MQSeries



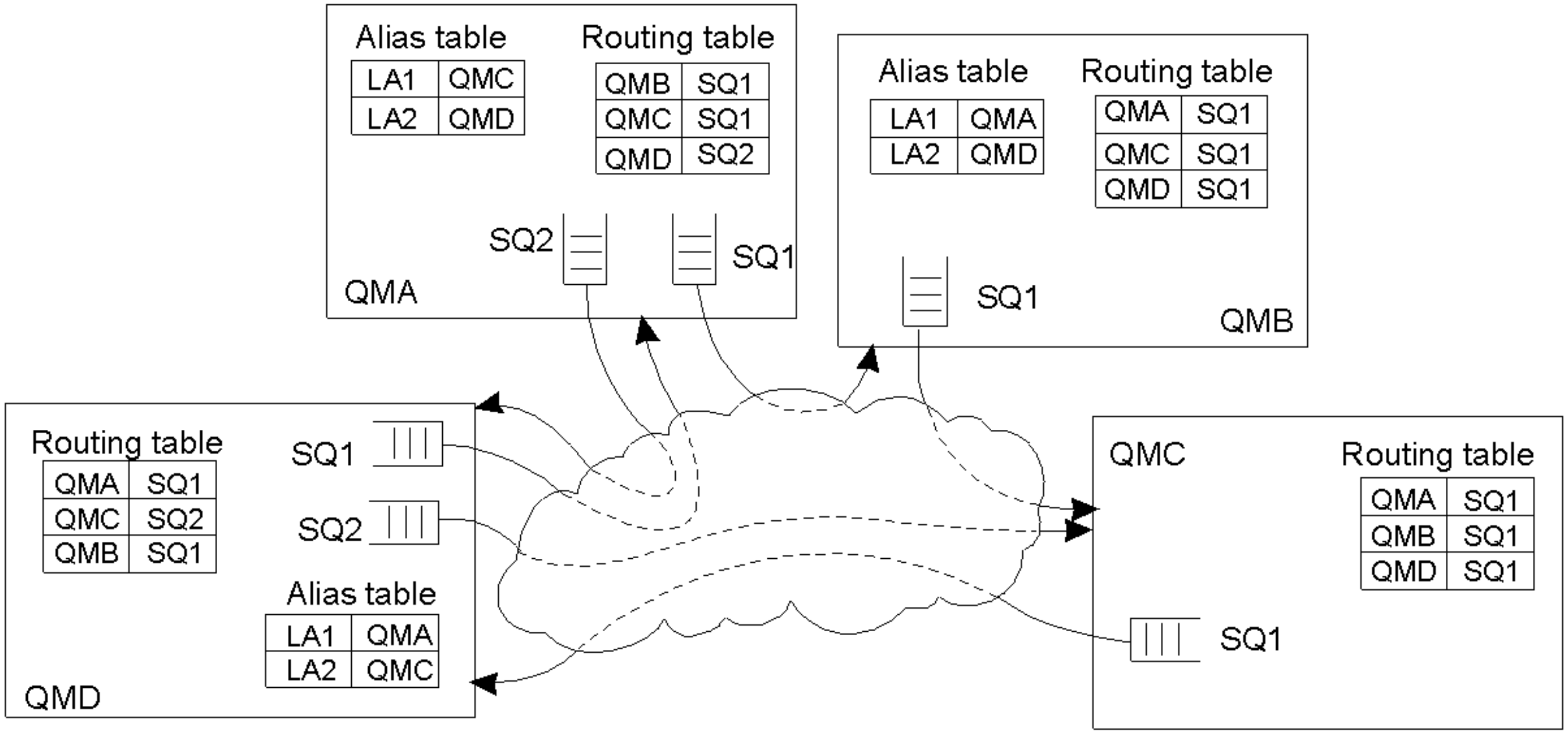
- Organization of IBM's MQSeries message-queuing system

Channels

Attribute	Description
Transport type	Determines the transport protocol to be used
FIFO delivery	Indicates that messages are to be delivered in the order they are sent
Message length	Maximum length of a single message
Setup retry count	Specifies maximum number of retries to start up the remote MCA
Delivery retries	Maximum times MCA will try to put received message into queue

- Attributes associated with message channel agents

Message Transfer (1)



- General organization of an MQSeries queuing network using routing tables and aliases.

Message Transfer (2)

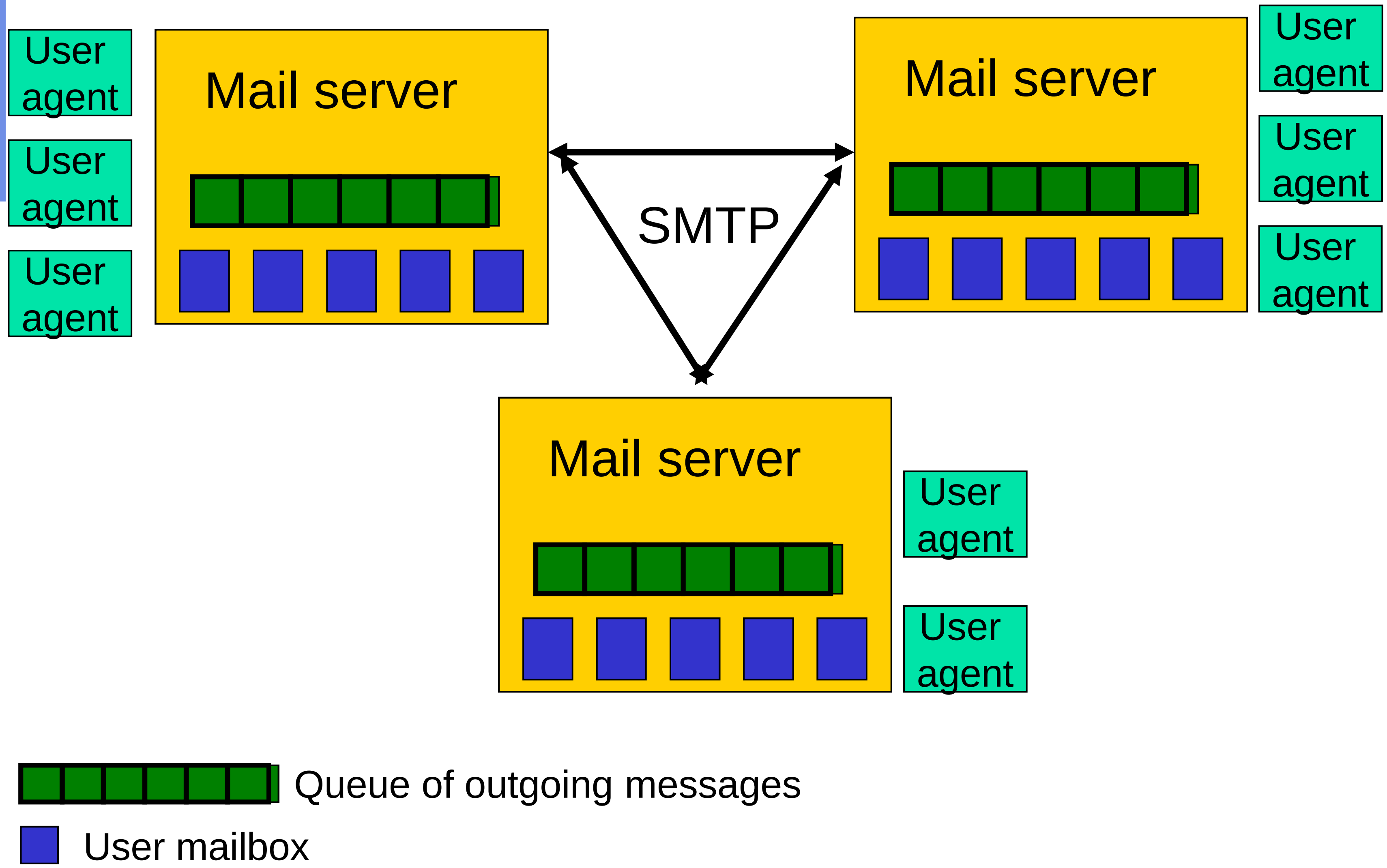
Primitive	**Description**
MQopen	Open a (possibly remote) queue
MQclose	Close a queue
MQput	Put a message into an opened queue
MQget	Get a message from a (local) queue

- Primitives available in an IBM MQSeries MQI

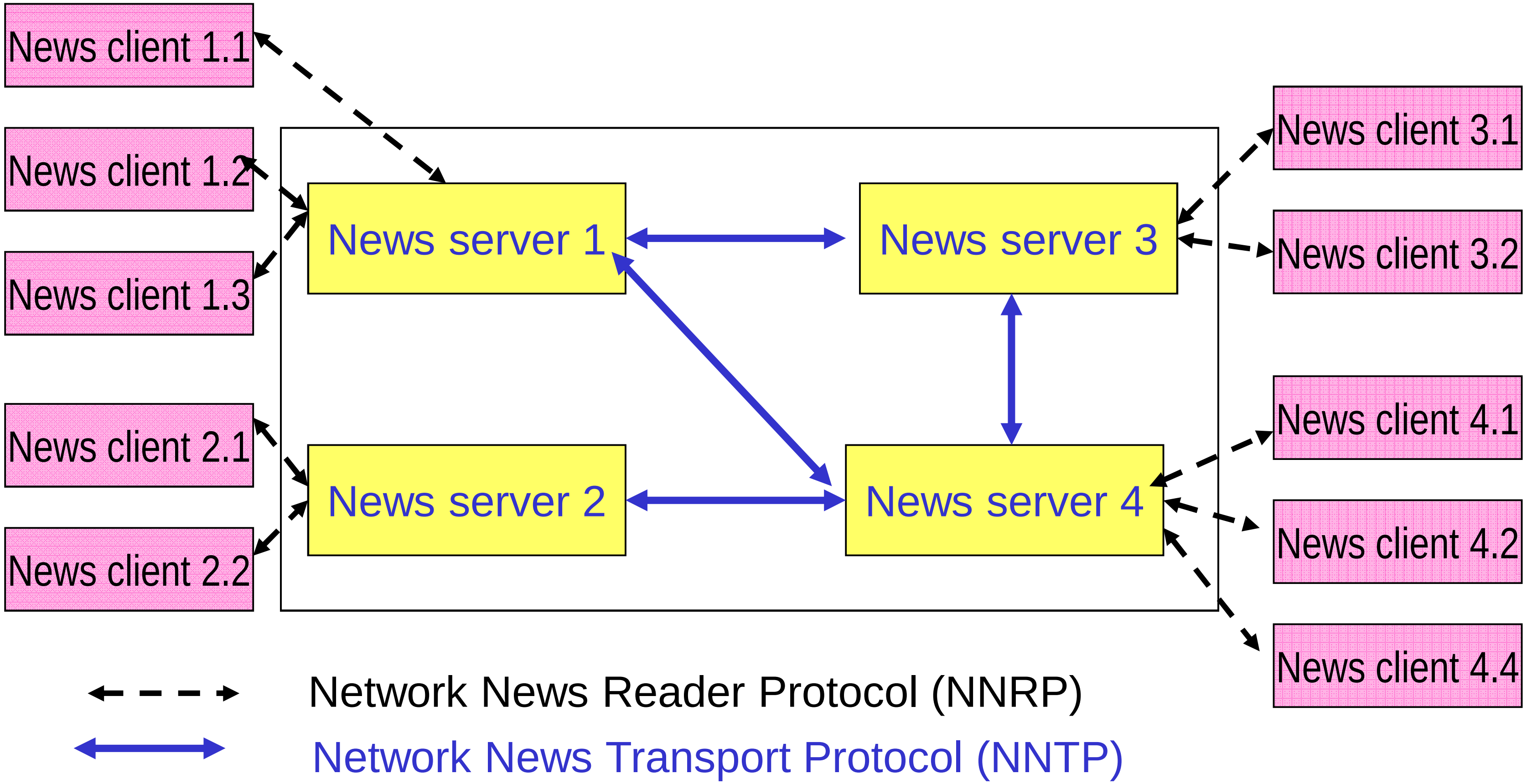
Simple Mail Transfer Protocol (SMTP)

- Processes
  - User agents (mail readers)
    - Eudora, pine, elm, outlook, messenger
  - Mail servers
    - Store messages
- SMTP
  - Uses TCP/IP
  - Uses DNS
- Client-to-server protocols
  - Pop (post office protocol)
  - Imap (internet mail access protocol)

SMTP


User agent
User agent
User agent
Mail server
SMTP
Mail server
User agent
User agent
User agent
Mail server
User agent
User agent
Queue of outgoing messages
User mailbox

Network News Protocol



- Uses TCP
- Servers “flood fill” their peers with new postings

Communication Endpoints

Communication Endpoints

- Identified via a DS wide unique identifier
  - Location transparency if ∃ global name scheme
- Different types of “receiving instances”:
  - Procedure or method, i.e. something passive to be invoked on the remote side
  - Process or thread receiving the message, i.e. something active on the remote side
  - Port handovers message to its owner process
  - Mailbox buffers message for its attached processes

Types of Service Instances

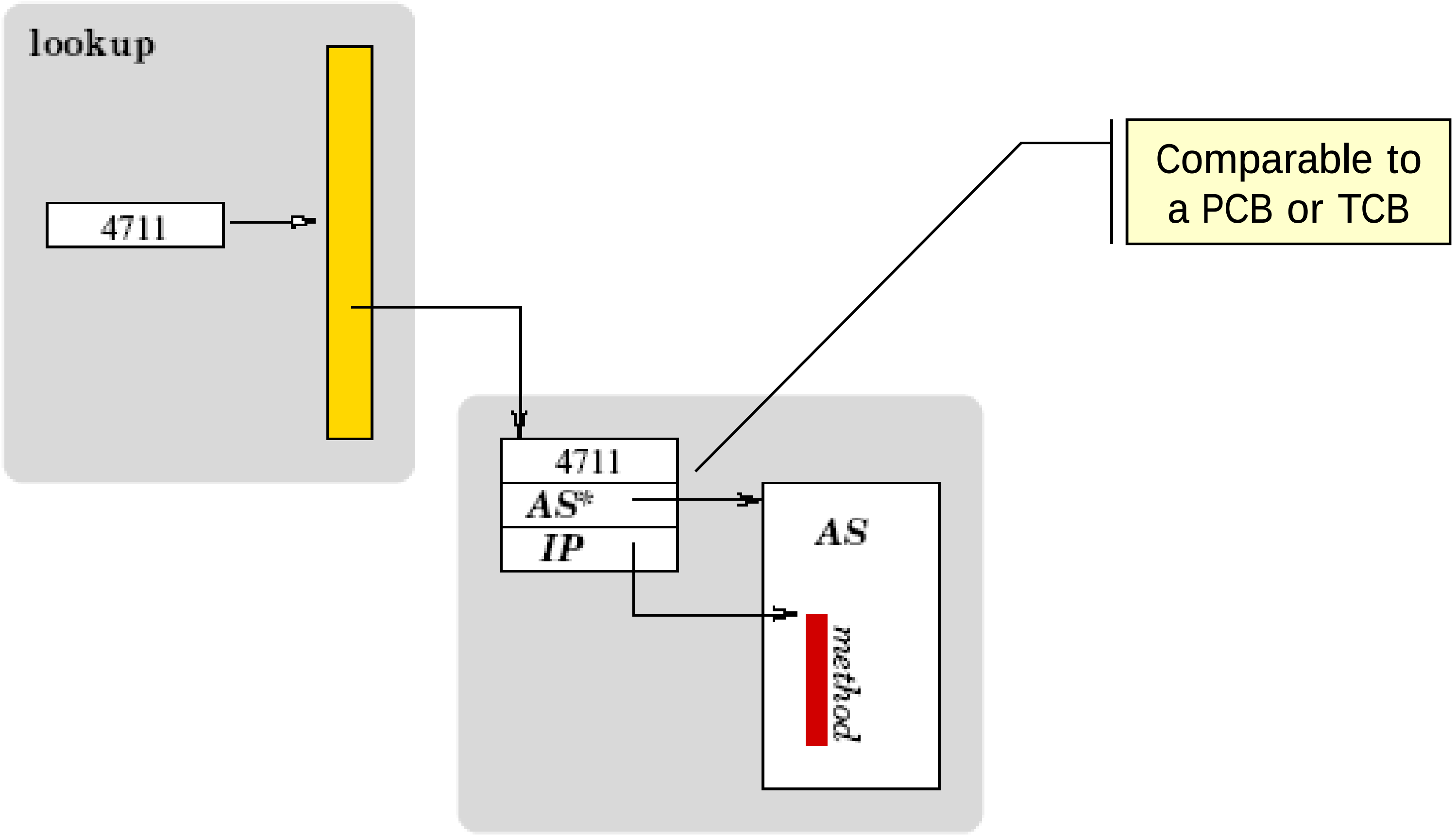
	Common AS[1]	Separate ASes
Passive Instance	Library Local Procedure Local Object	Remote Procedure Remote Object
Active Instance	Thread	Process Task

[1] except SASOS

Communication Endpoints

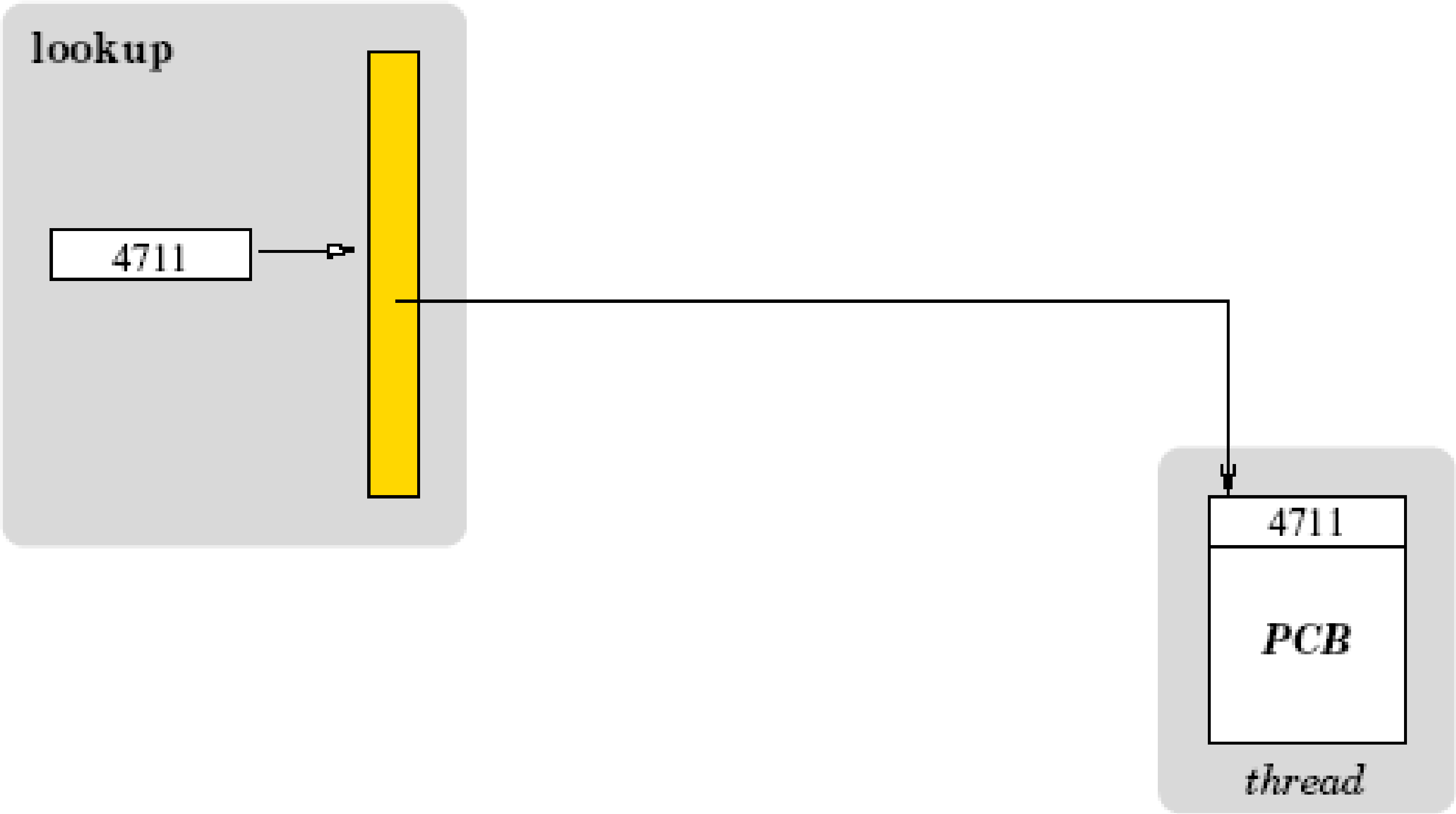
- Identifier of the place of destination can have
  - meaning that is location transparent (or not)
  - ∃ tradeoff between
    - performance and
    - flexibility or transparency
  - Value of identifier must be system wide unique (at least for a while)

Example Remote Procedure

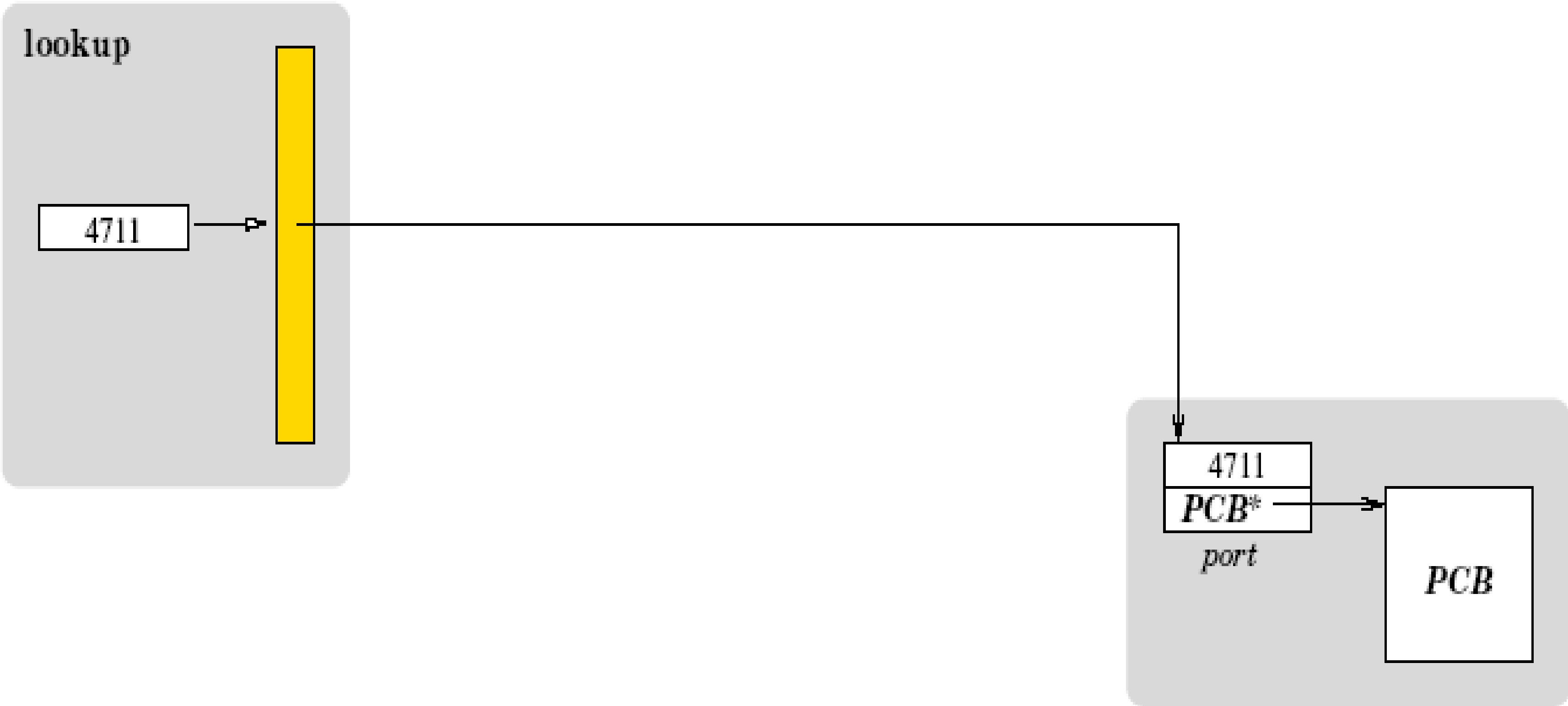


• If caller must know about the node hosting the remote procedure we have no location transparency

Example Process

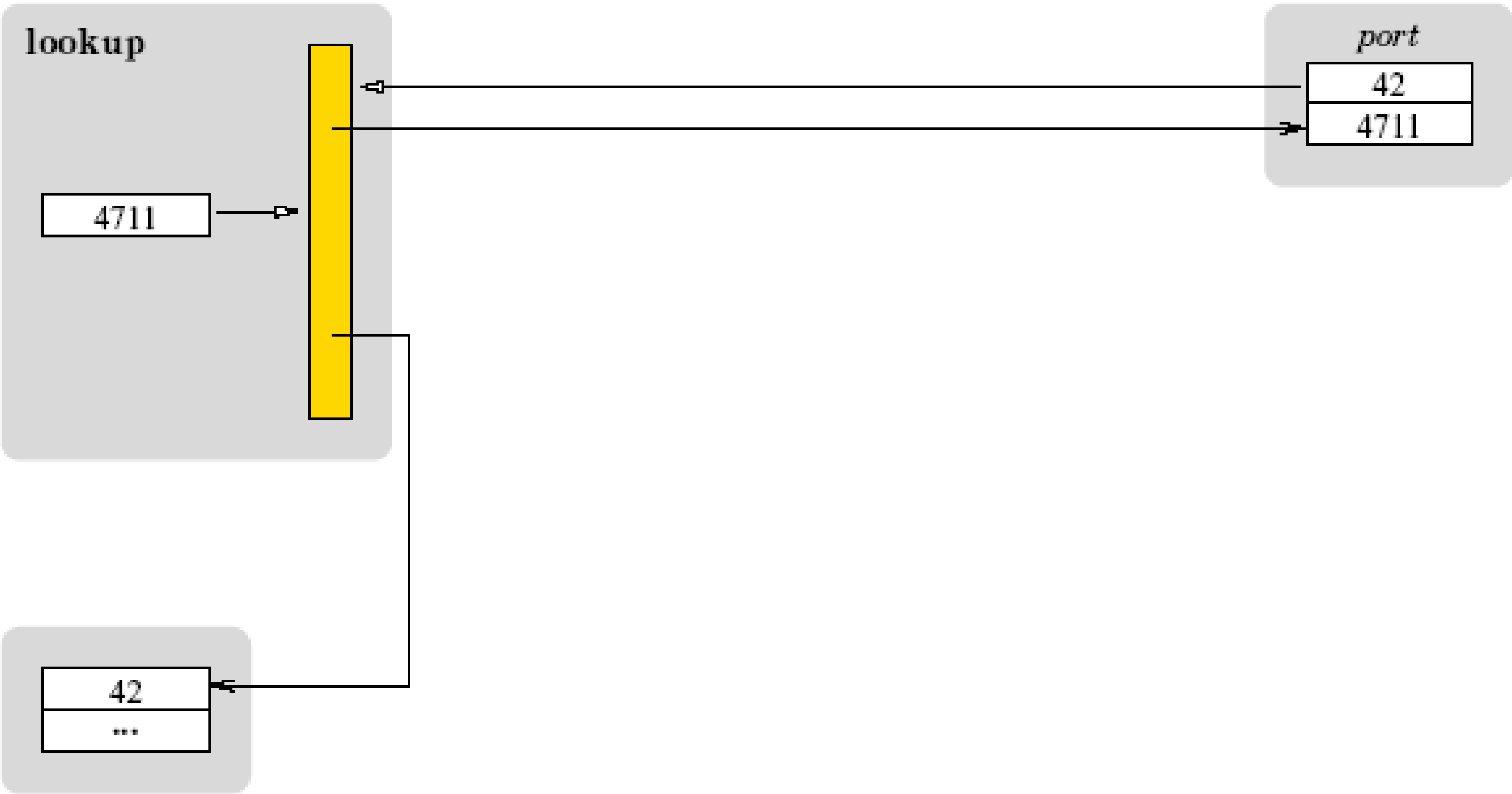


Example Port

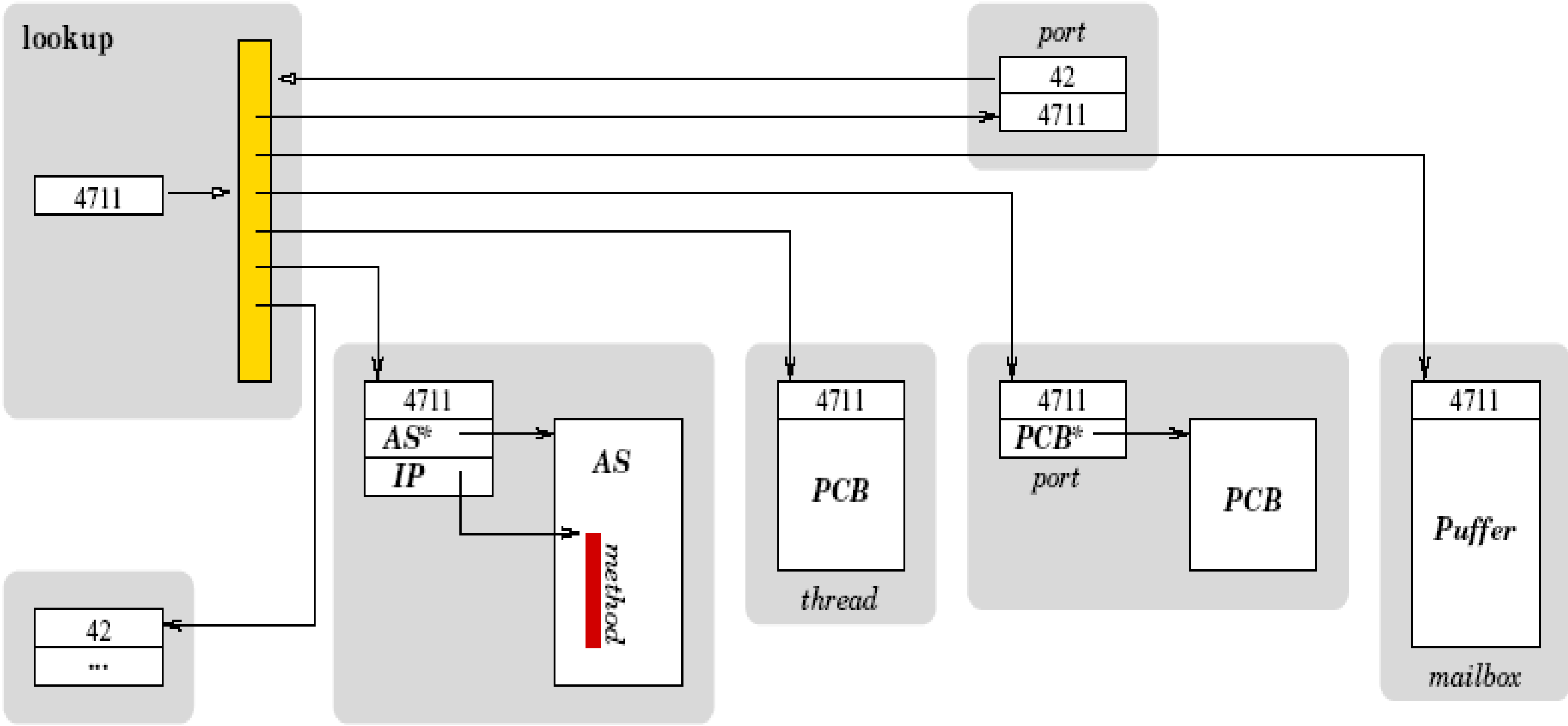


Example Mailbox

Example Port Chain



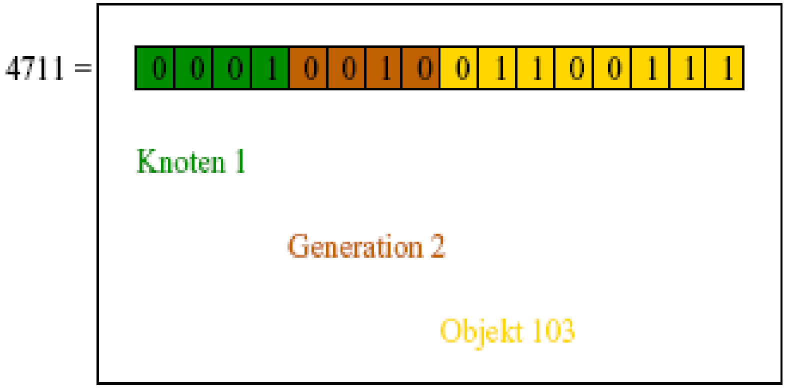
Summary: Endpoints

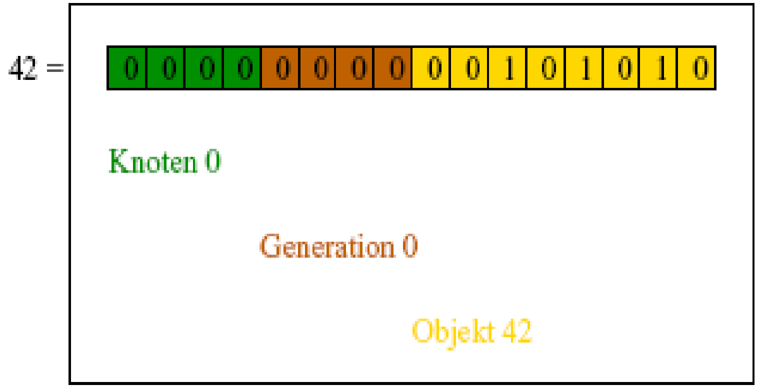


How to achieve Uniqueness?

- Identifier must have a value excluding ambiguity
  - Random number depends on quality of random number generator
  - Time stamp needs a global time
  - Processor number depends on manufacturer
- Already locally you need uniqueness, e.g.
  - Used only once (see: UNIX PID)
  - Generation number differs between reused identifiers
  - Address of an object in RAM
- Degree of uniqueness depends on potential range of values
  - See Y2K problems

Structured Identifiers



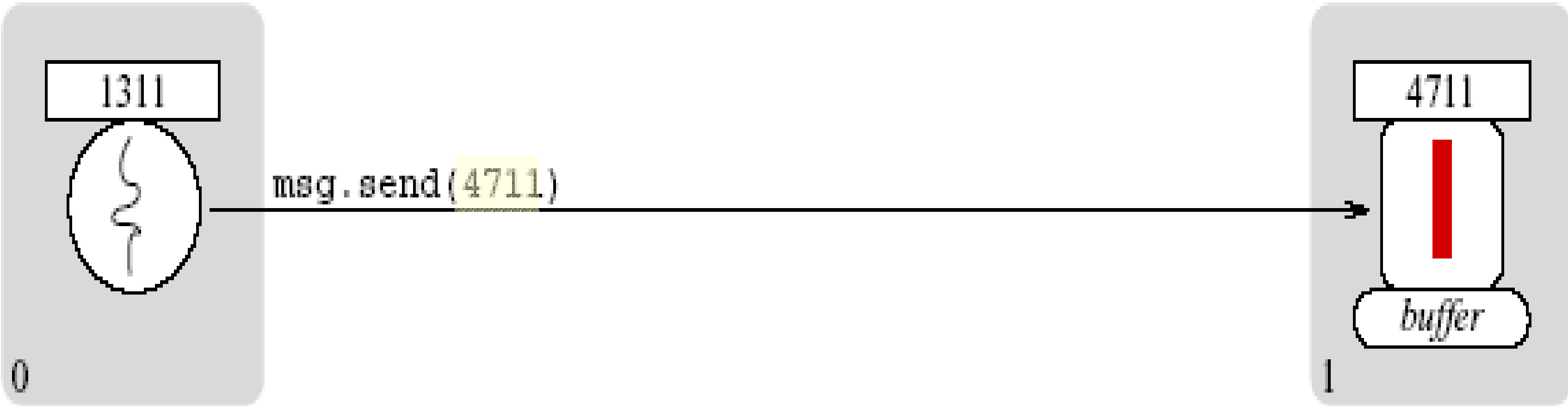


- Typically the structure of an identifier is not visible from outside
  - It enhances an efficient lookup for an insider
- Identifier is location transparent (but contains location hints)

Communication Process

- Direct: identifier is a PID, i.e. either a process or a procedure or a TID, i.e. a thread-id
  - Messages are sent to the corresponding AS
- Indirect: identifier is a port-id or a mailbox-id
  - Messages are sent to receiver via a port or mailbox
- Connection oriented: identifier is a port-id
  - Connection exists between ports
  - Using a connection helps to reserve resources

Direct Communication (1)

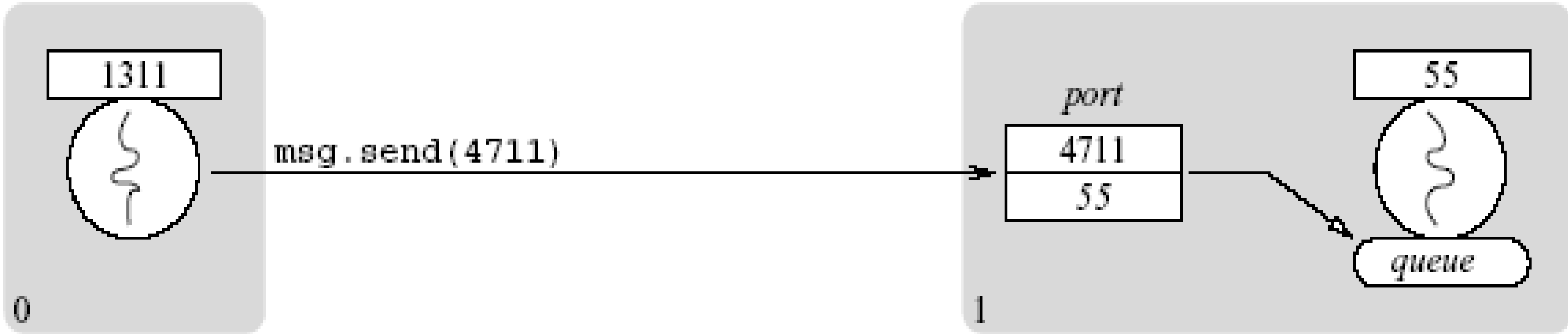


- Address a remote procedure to do work specified in the request message
- Scheme neither supports migration nor failure transparence

Direct Communication (2)

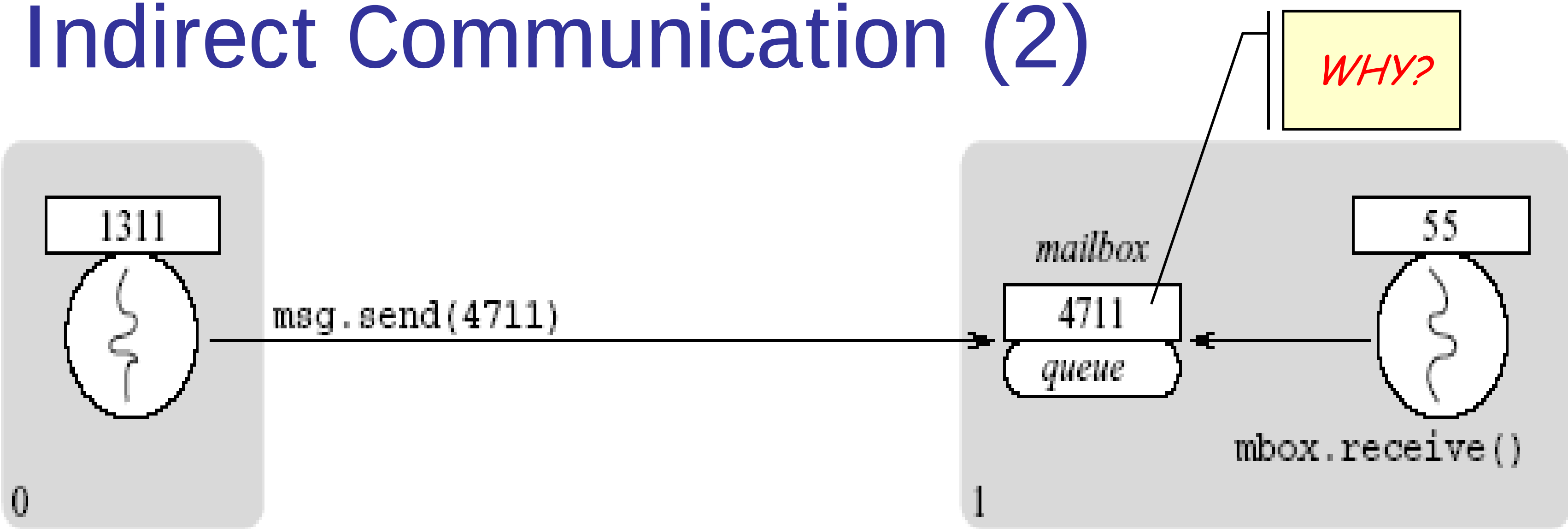
- Address a remote process to do work specified in the request message
  - Time when this message will received determines the scheduling of the related worker thread (process)
  - Potential scheduling latency requires a message queue
- Scheme neither supports migration nor failure transparency

Indirect Communication (1)



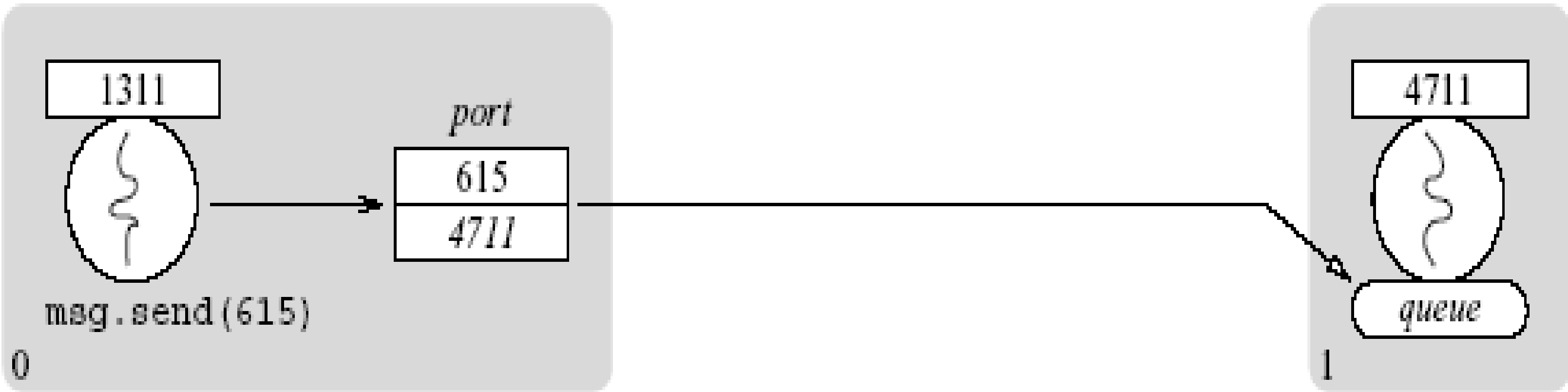
- Address a remote port to forward the request message
  - Receiver (with PID 55) is loosely coupled (docked) with the entrance port
  - Binding is dynamic and can relate to multiple ports
- Scheme supports migration, but not failure transparency

Indirect Communication (2)



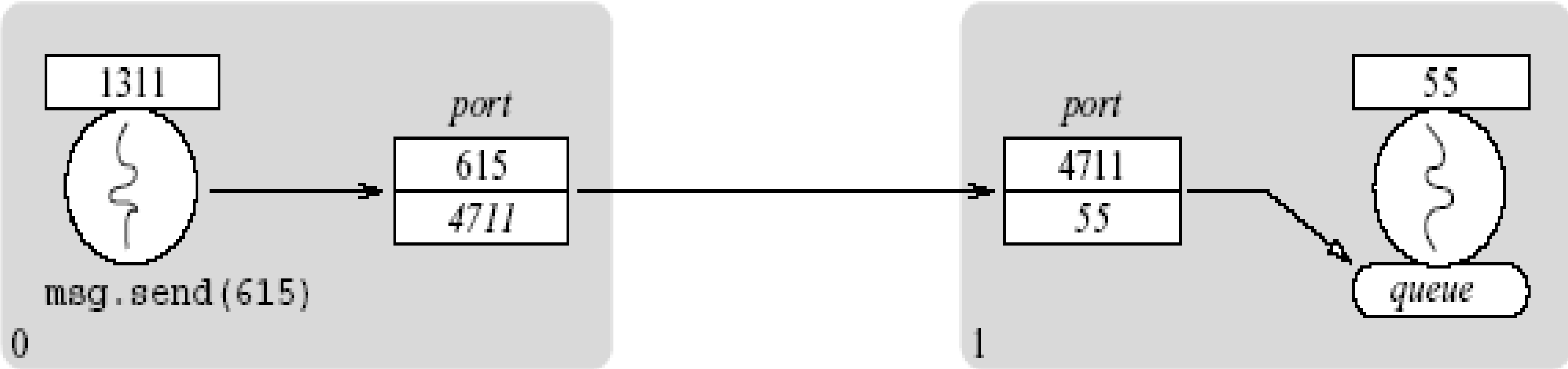
- You address a remote mailbox to queue messages
  - Multiple processes can receive from the same mailbox
  - Typical for multithreaded server
- Scheme neither supports migration nor failure transparency

Connection Oriented Communication (1)



- You address the local port to forward your request message
  - Linking between send-port and remote process is dynamic
  - Identifier of the remote process can be a replicate
- Scheme supports migration and failure transparency

Connection Oriented Communication (2)



- Address local port to forward request message via a port chain
  - Dynamic linking of send-port to receive-port either according to 1:1 or N:1
  - In case of node crashes you have to repair broken port-chains
- Scheme supports migration and failure transparency