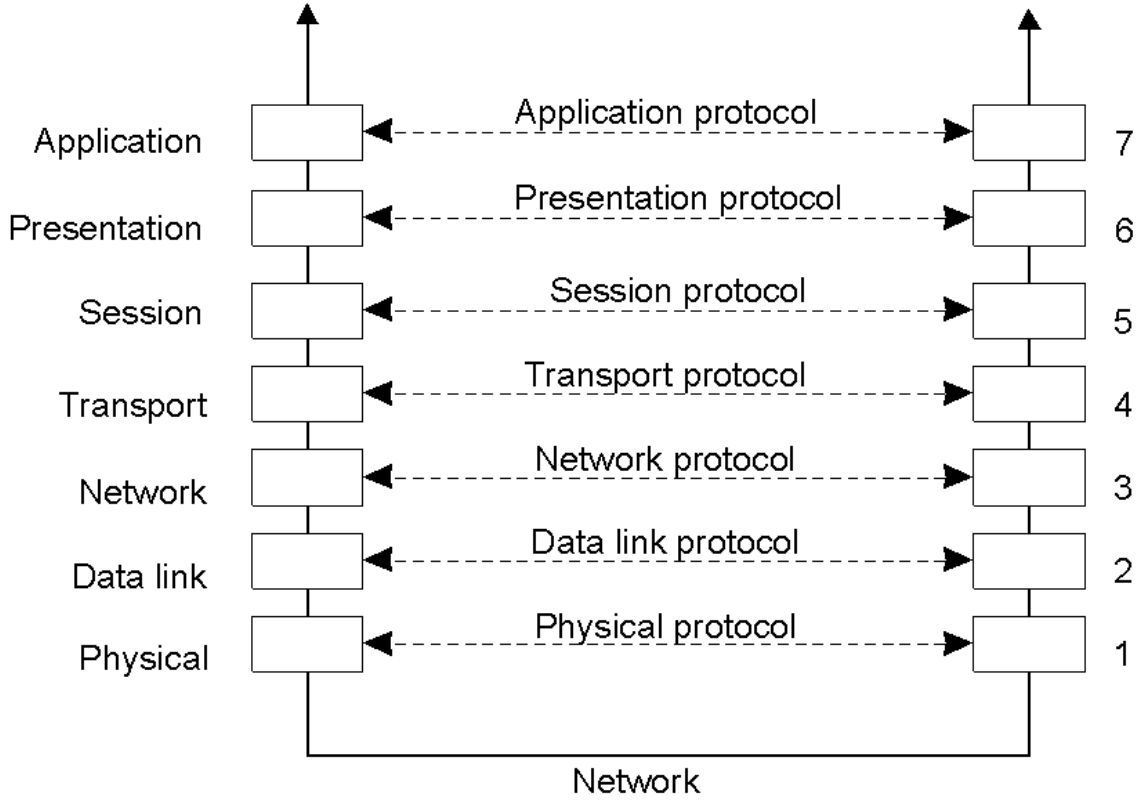
Communication

Inter-Process Communication (IPC)

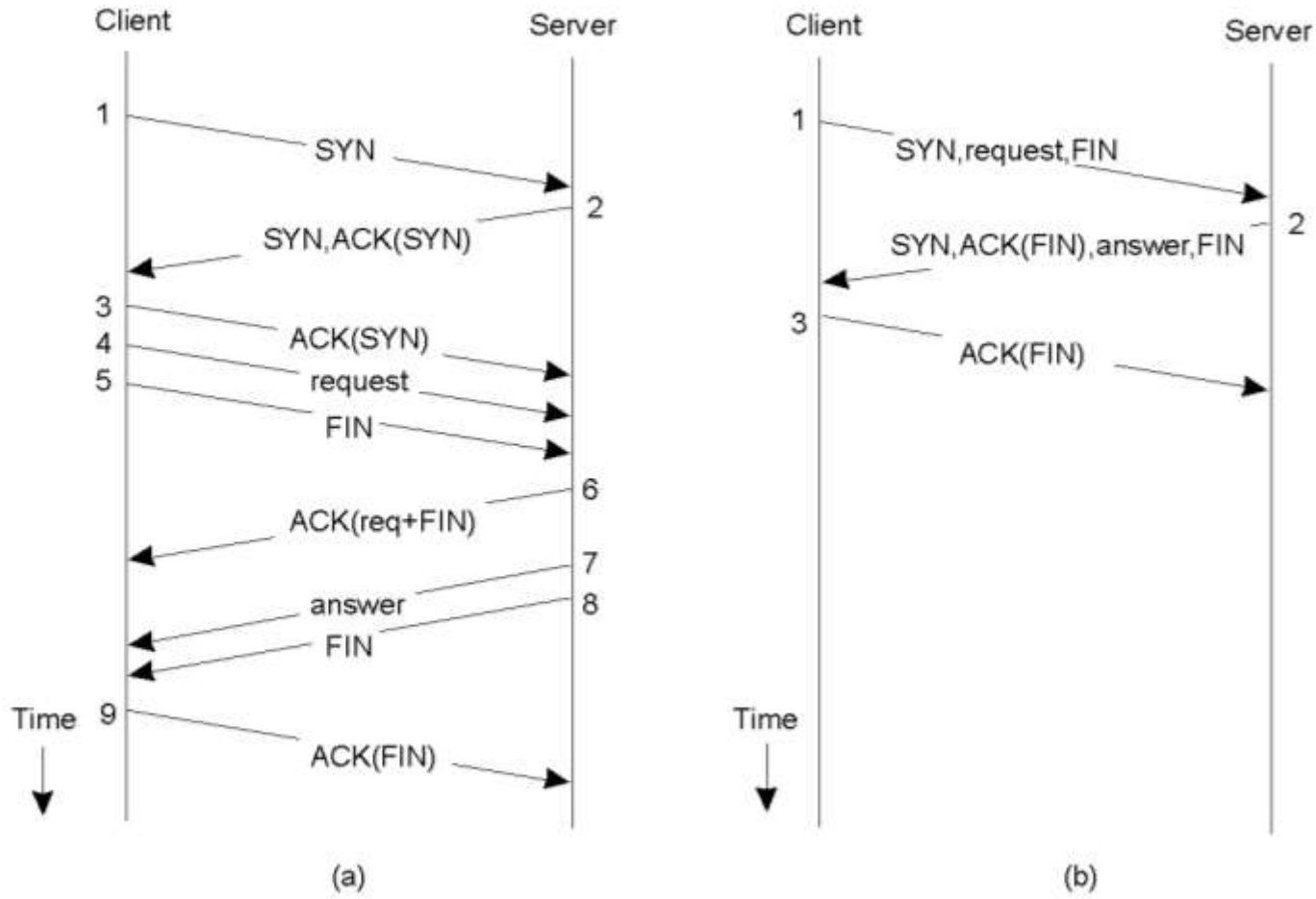
- The “heart” of every distributed system.
- Question: how do processes on different machines exchange information?
  - Answer: with difficulty … ☹
- Established computer network facilities are **too primitive**:
  - resulting in DSs that are too difficult to develop – new model is required
- Four IPC models are popular:
    - RPC; RMI; MOM and Streams

x-Layered Protocols



Layers, interfaces, and protocols in the OSI model.

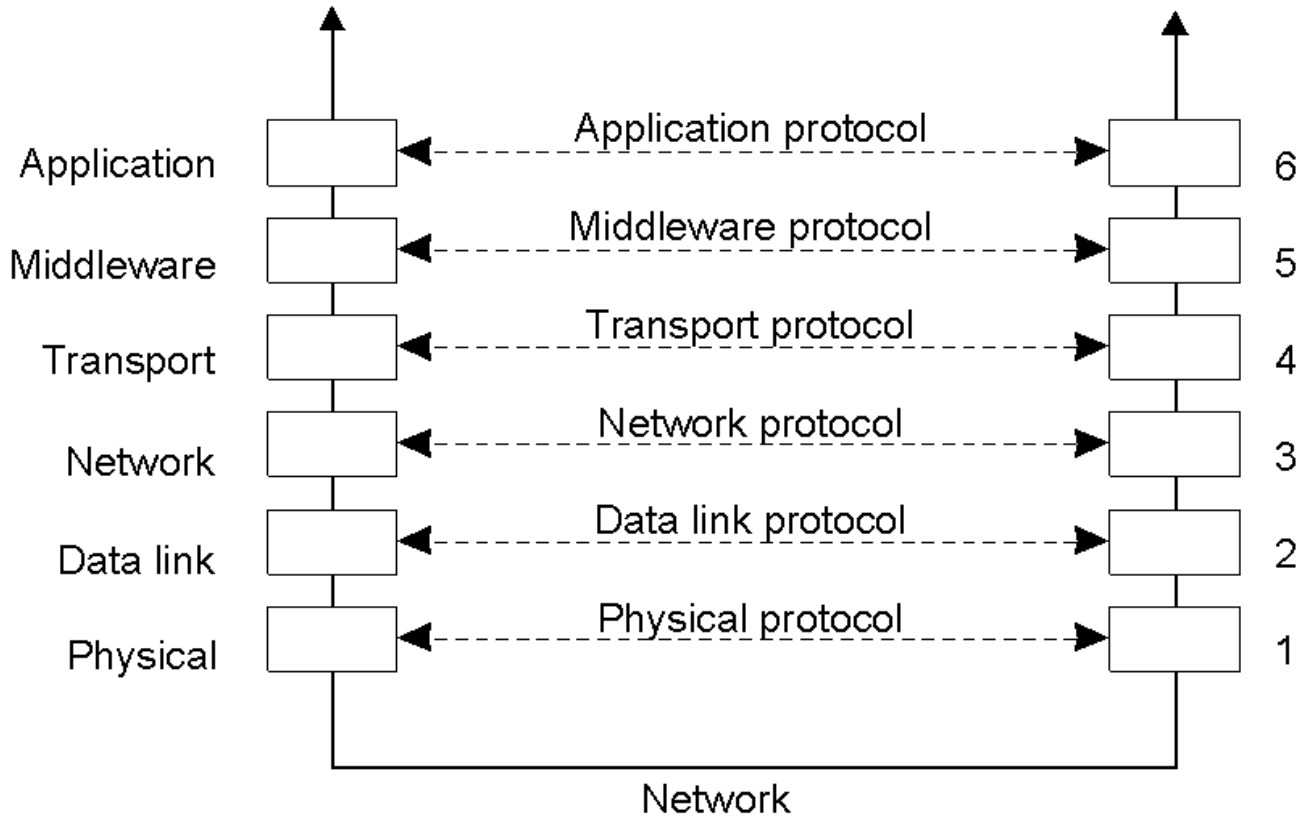
Client-Server TCP



a) Normal operation of TCP.

b) Transactional TCP.

x-Middleware Protocols



An adapted reference model for networked communication.

RPC: Remote Procedure Call

Birrell and Nelson (1984)

"To allow programs to call procedures located on other machines."

Effectively removing the need for the Distributed Systems programmer to worry about all the details of network programming (i.e. *no more sockets*).

Conceptually simple, but …

Complications: More on RPC

- Two machine architectures may not (or need not) be identical.
- Each machine can have a different address space.
- How are parameters (of different and complex types) passed to/from a remote procedure?
- What happens if any DS machine crashes while the procedure is being called?

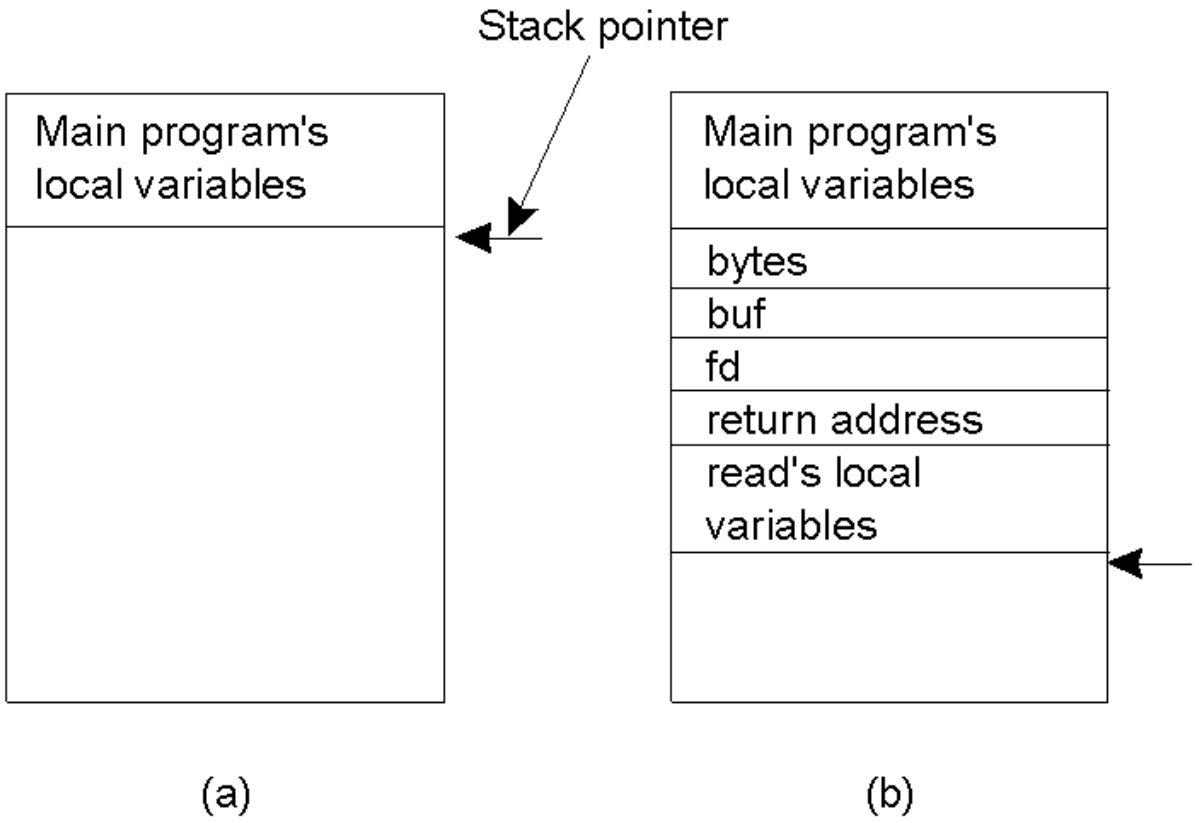
How RPC Works: Part 1

As far as the programmer is concerned, a “remote” procedure call looks and works identically to a “local” procedure call.

In this way, **transparency** is achieved.

Before looking into RPC in action, let’s consider a conventional “local” procedure call.

Conventional "Local" Procedure Call



a) Parameter passing in a local procedure call: the stack **before** the call to read.

b) The stack **while** the called procedure is active.

How RPC Works: Part 2

The procedure is “split” into two parts:

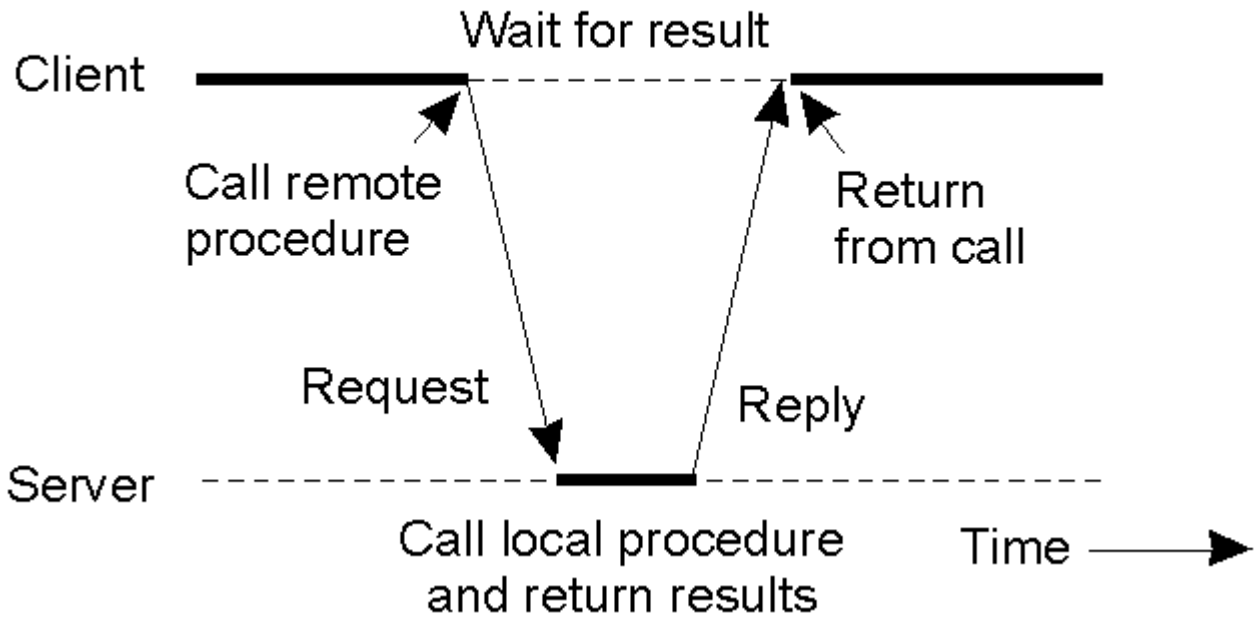
- The CLIENT “stub” – implements the interface on the local machine through which the remote functionality can be invoked,

and

- The SERVER “stub” – implements the actual functionality, i.e. does the real work!

Parameters are “**marshalled**” by the client prior to transmission to the server.

Client and Server "Stubs"

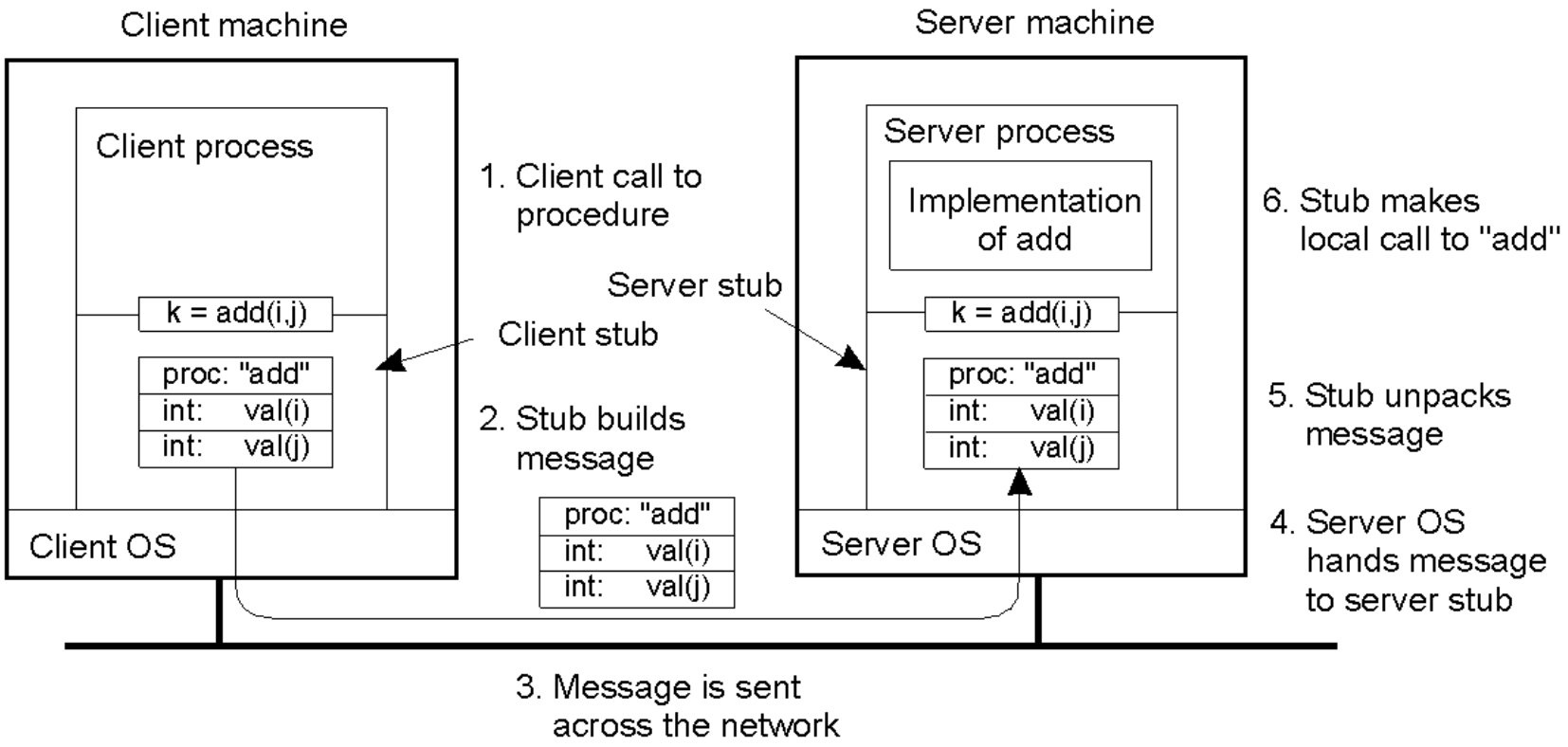


Principle of RPC between a client and server program – no big surprises here …

The Ten 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

Passing Value Parameters (1)



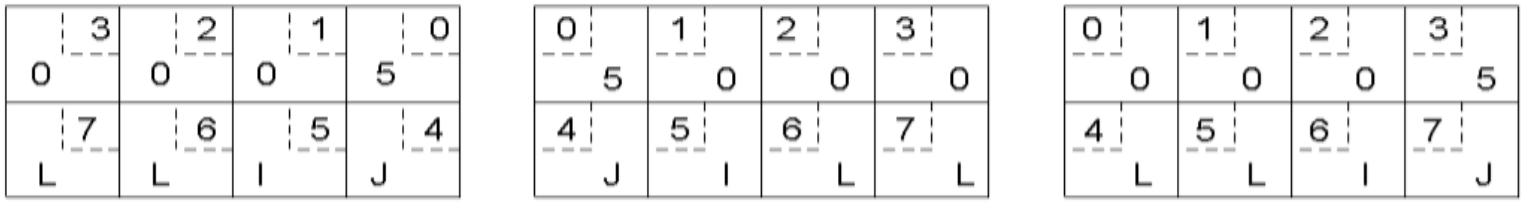
Steps involved in doing remote computation through RPC.

RPC Problems

- RPC works really well if all the machines are homogeneous.
- Complications arise when:
  - Machines use different character encodings, e.g. EBCDIC or ASCII.
  - Byte-ordering is also a problem:
    - Intel machines “big-endian” / Sun Sparc’s “little-endian”.

Extra mechanisms are required to be built into the RPC mechanism to provide for these types of situations – this adds **complexity**.

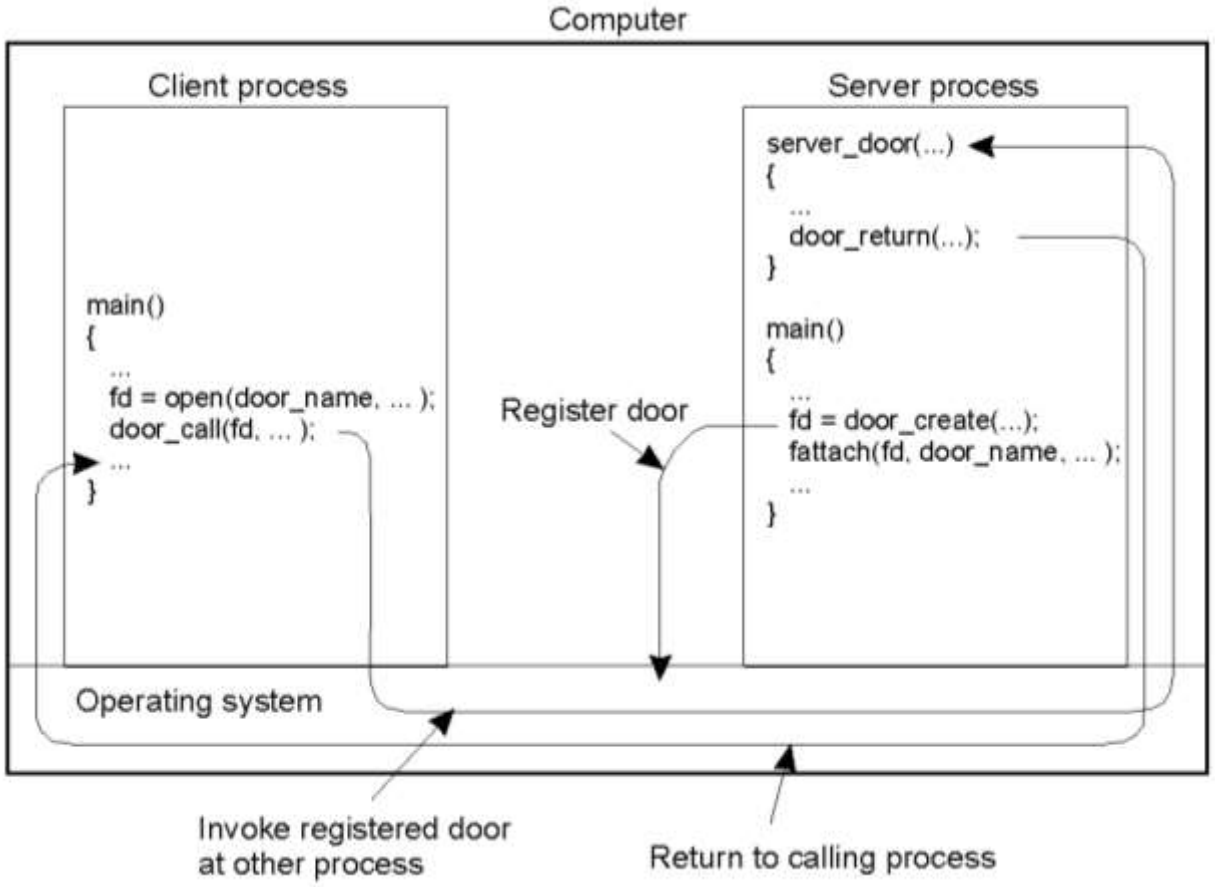
Passing Value Parameters (2)



a) Original message on the Pentium (Intel).
b) The message after receipt on the SPARC.
c) The message after being inverted – which still does not work properly.

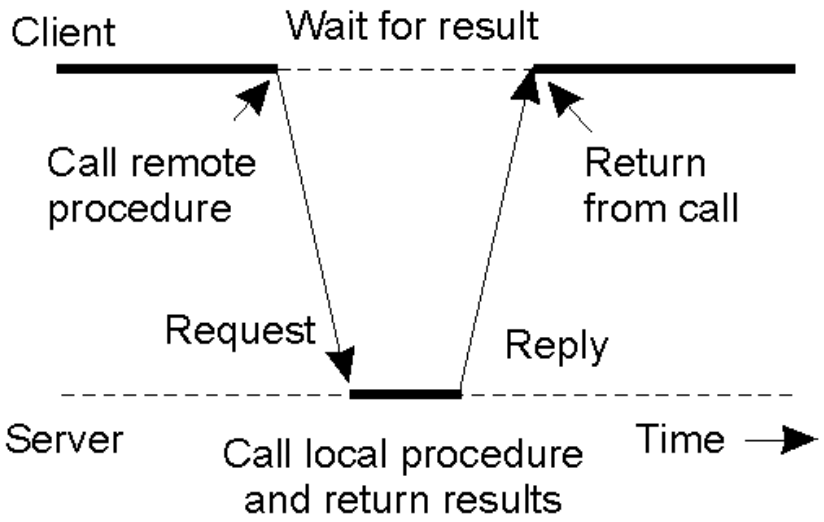
[The little numbers in boxes indicate the address of each byte]

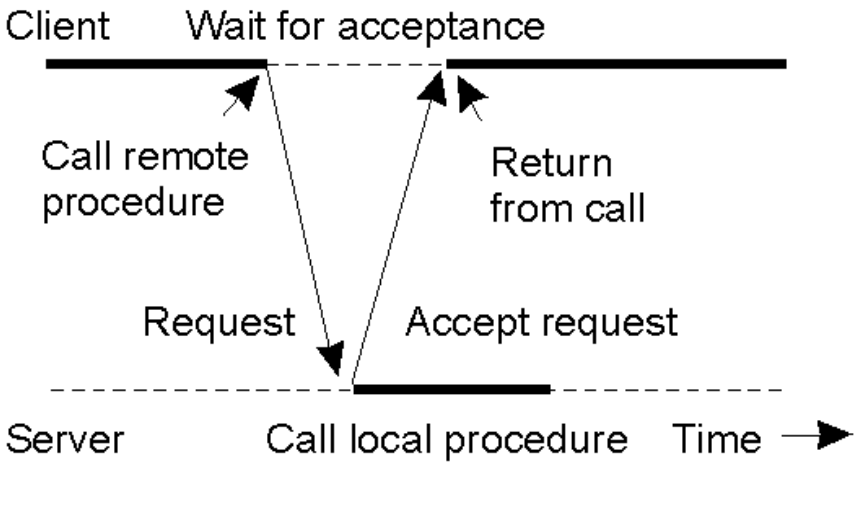
An RPC Variation: Doors



The principle of using doors as IPC mechanism:
Processes are co-located on a single machine (as RPC would be overkill).
Upside: a single mechanism support DS programming.
Downside: loss of transparency.

Asynchronous RPC

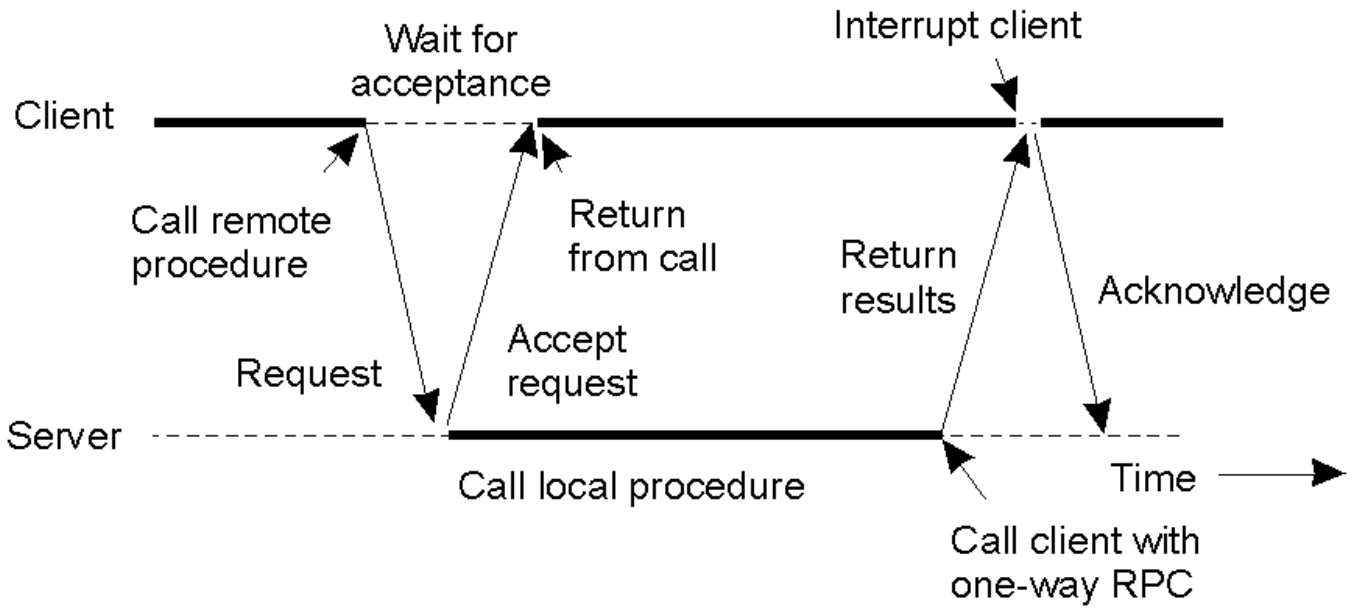




a) The interconnection between client and server in a traditional RPC (BLOCKING occurs – client waits for reply/return).

b) The interaction using asynchronous RPC (NON BLOCKING - useful when the client does not need or expect a result).

Asynchronous RPC: Deferred Synchronous RPC



A client and server interacting through two asynchronous RPCs – allows a client to perform other useful work while waiting for results.

Interface Definition Language (IDL)

RPCs typically require development of custom protocol interfaces to be effective.

Protocol interfaces are described by means of an Interface Definition Language (IDL).

IDLs are “language-neutral” – they do not presuppose the use of any one programming language.

That said, most IDLs look a lot like C …

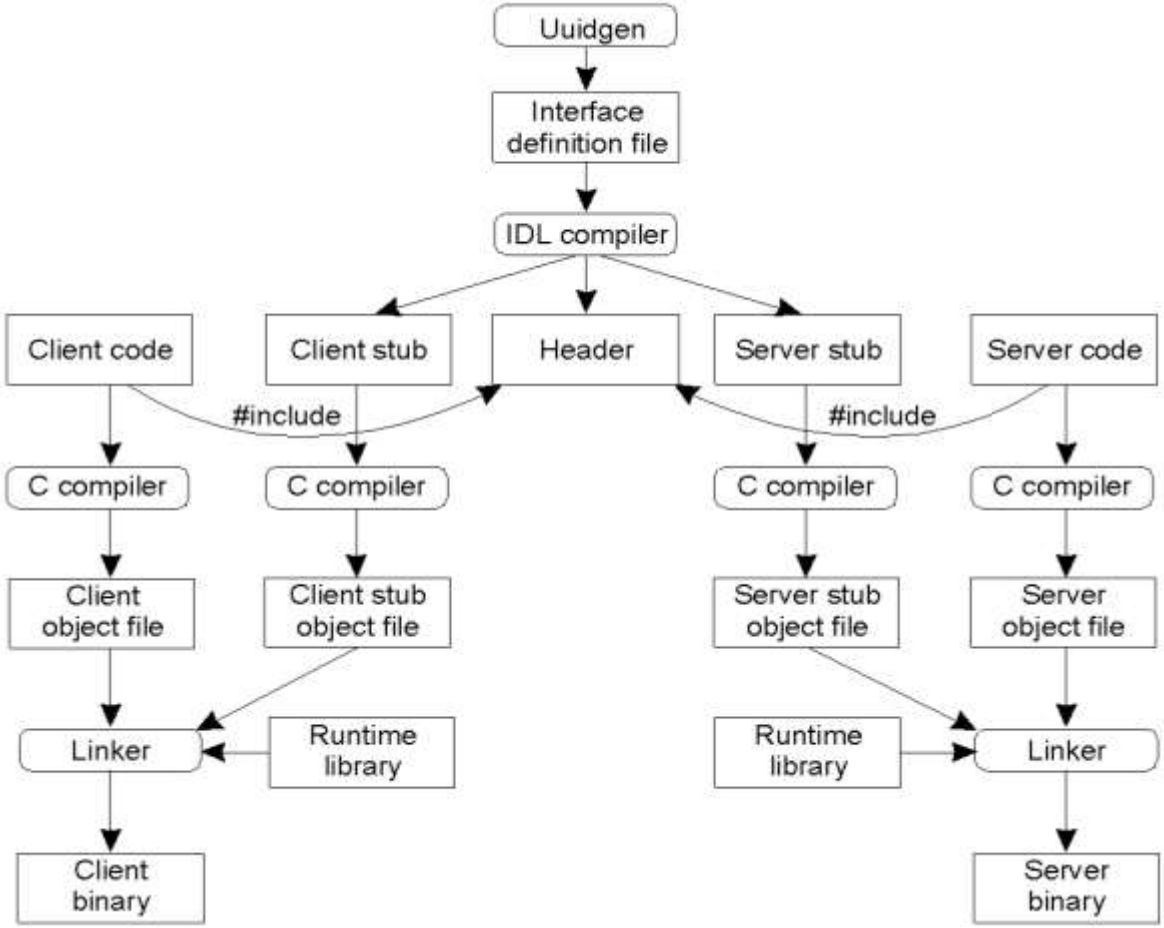
DCE: An Example RPC

The Open Group's standard RPC mechanism.

In addition (to RPC) DEC provides:

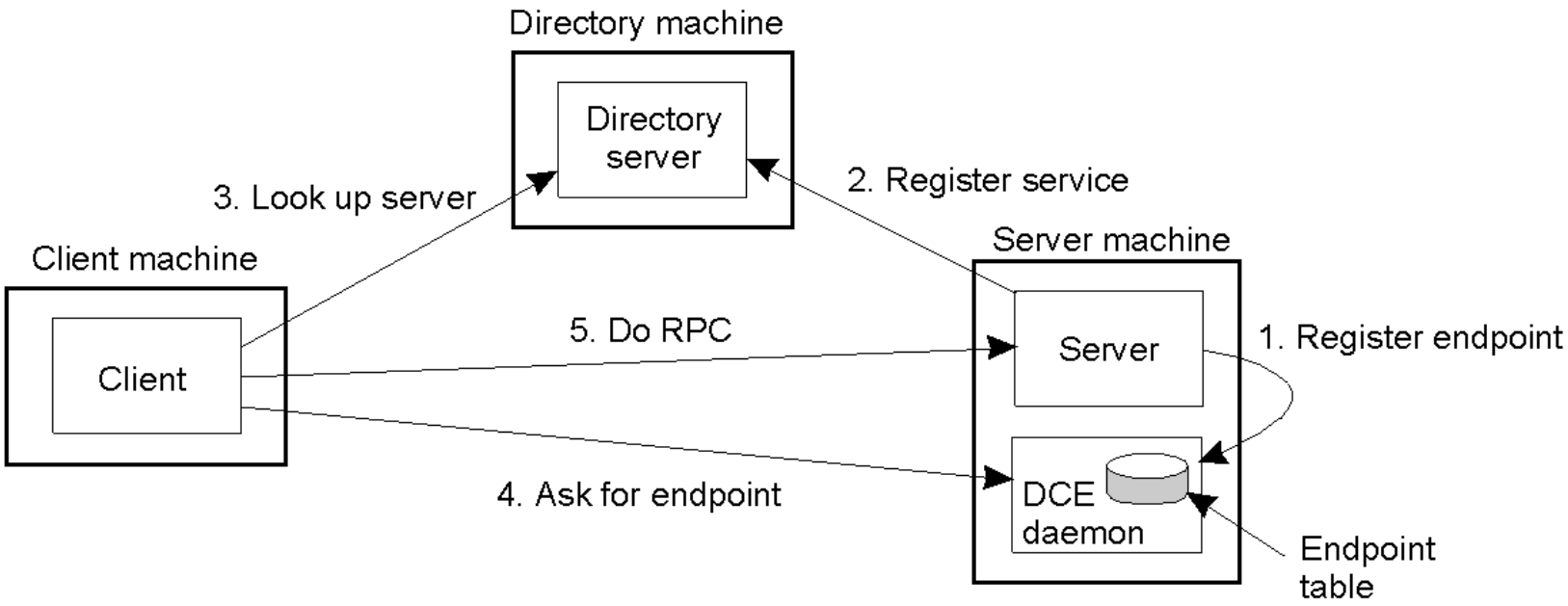
- Distributed File Service.
- Directory Service (name lookups).
- Security Service.
- Distributed Time Service.

DCE: Writing a Client and a Server



The steps in writing a client and a server in DCE RPC.

DCE: “Binding” a Client to a Server



Client-to-server binding in DCE.

A “directory service” provides a way for the client to look-up the server.

RPC Summary

The DS de-facto standard for communication and application distribution (at the procedure level).

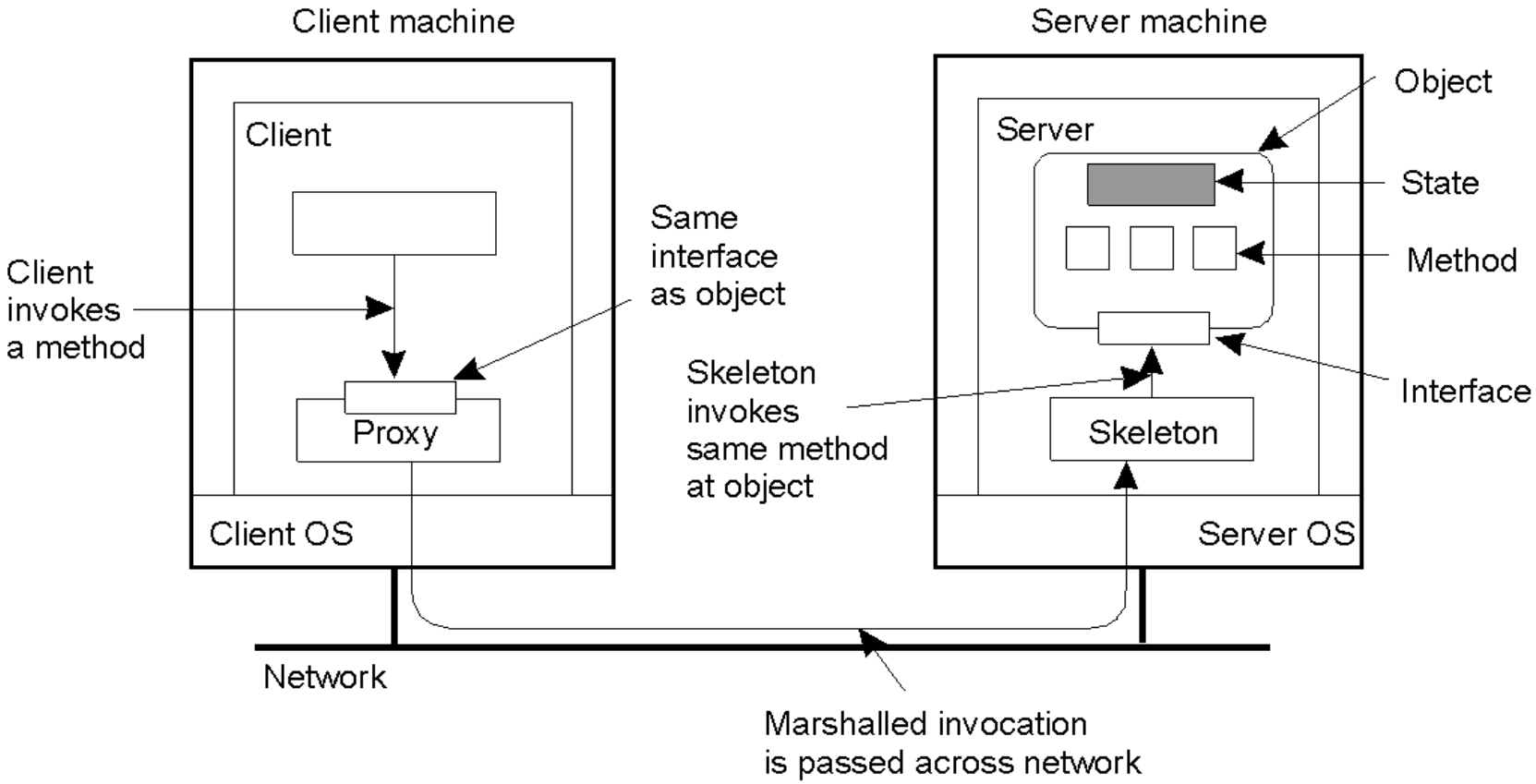
- It is mature.
- It is well understood.
- It works!

RMI: Remote Method Invocation

- “Remote objects” can be thought of as an expansion of the RPC mechanism (to support OO systems).
- An important aspect of objects is the definition of a well-defined interface to “hidden” functionality.
- Method calls support state changes within the object through the defined interface.
- An object may offer multiple interfaces.
- An interface may be implemented by multiple objects.

- Within a DS, the object interface resides on one machine, and the object implementation resides on another.

The Distributed Object



Common organization of a remote object with client-side “proxy”.

- The “proxy” can be thought of as the “client stub”.
- The “skeleton” can be thought of as the “server stub”.

Compile-time vs. Run-time Objects

- Compile-time distributed objects generally assume the use of a particular programming language (Java or C++).
- This is often seen as a drawback (inflexible).

- Run-time distributed objects provide *object adaptors* to objects, designed to remove the compile-time programming language restriction.
- Using **object adaptors** allows an object implementation to be developed in any way – as long as the resulting implementation "appears" like an object, then things are assumed to be OK.

Persistent vs. Transient Objects

- A “persistent” distributed object continues to exist even after it no longer exists in the address space of the server.
- It is stored (perhaps on secondary storage) and can be re-instantiated at a later date by a newer server process (i.e. by a newer object).

- A “transient” distributed object does not persist.
- As soon as the server exits, the transient object is destroyed.

Which one is better is the subject of much controversy …

Static vs. Dynamic Invocation

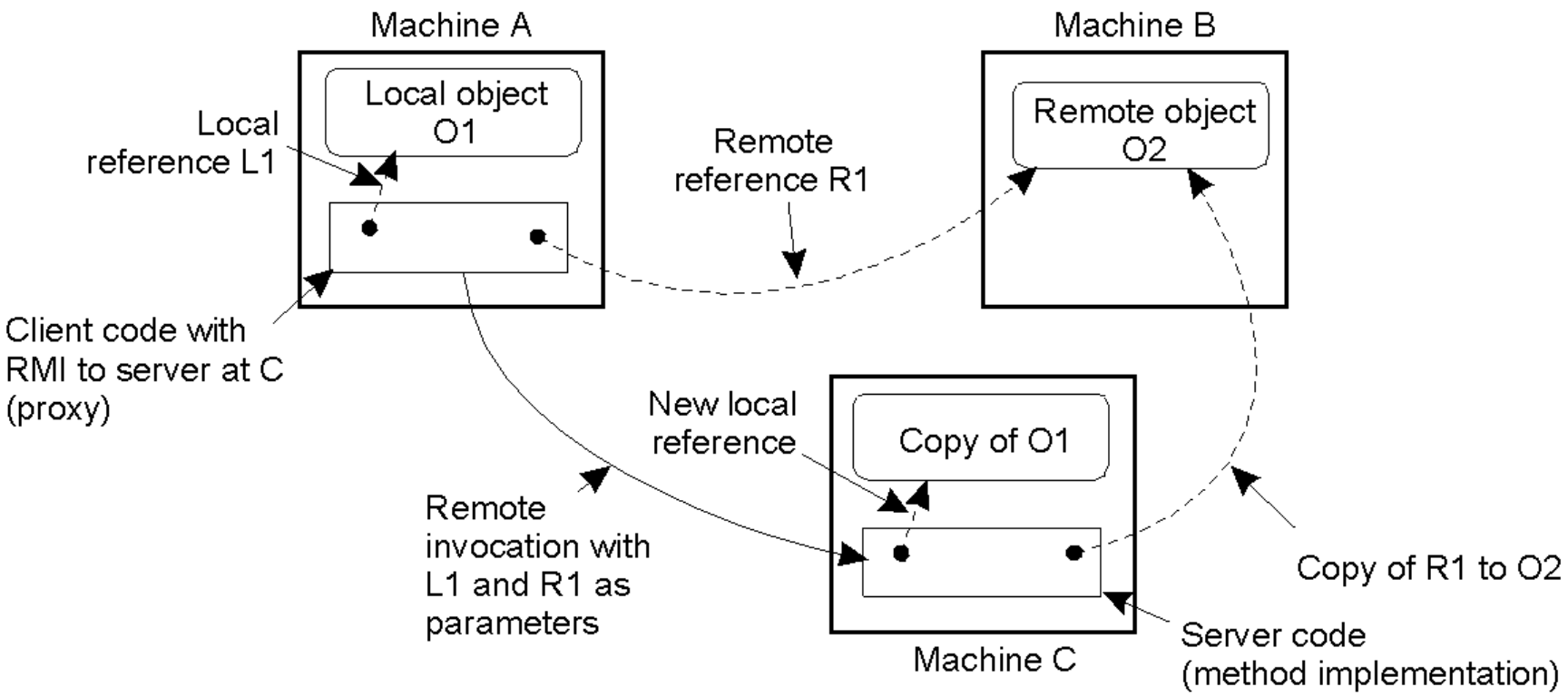
Predefined interface definitions support *static invocation*:

- all interfaces are known up-front.
- changes to the interface requires all applications (i.e. clients) to be recompiled.

Dynamic invocation "composes" a method at run-time

- interface can "come and go" as required.
- interfaces can be changed without forcing a recompile of client applications.

Remote Objects: Parameter Passing



The situation when passing an object by reference or by value.

Note: O1 is passed by value; O2 is passed by reference.

Example: DCE Remote Objects

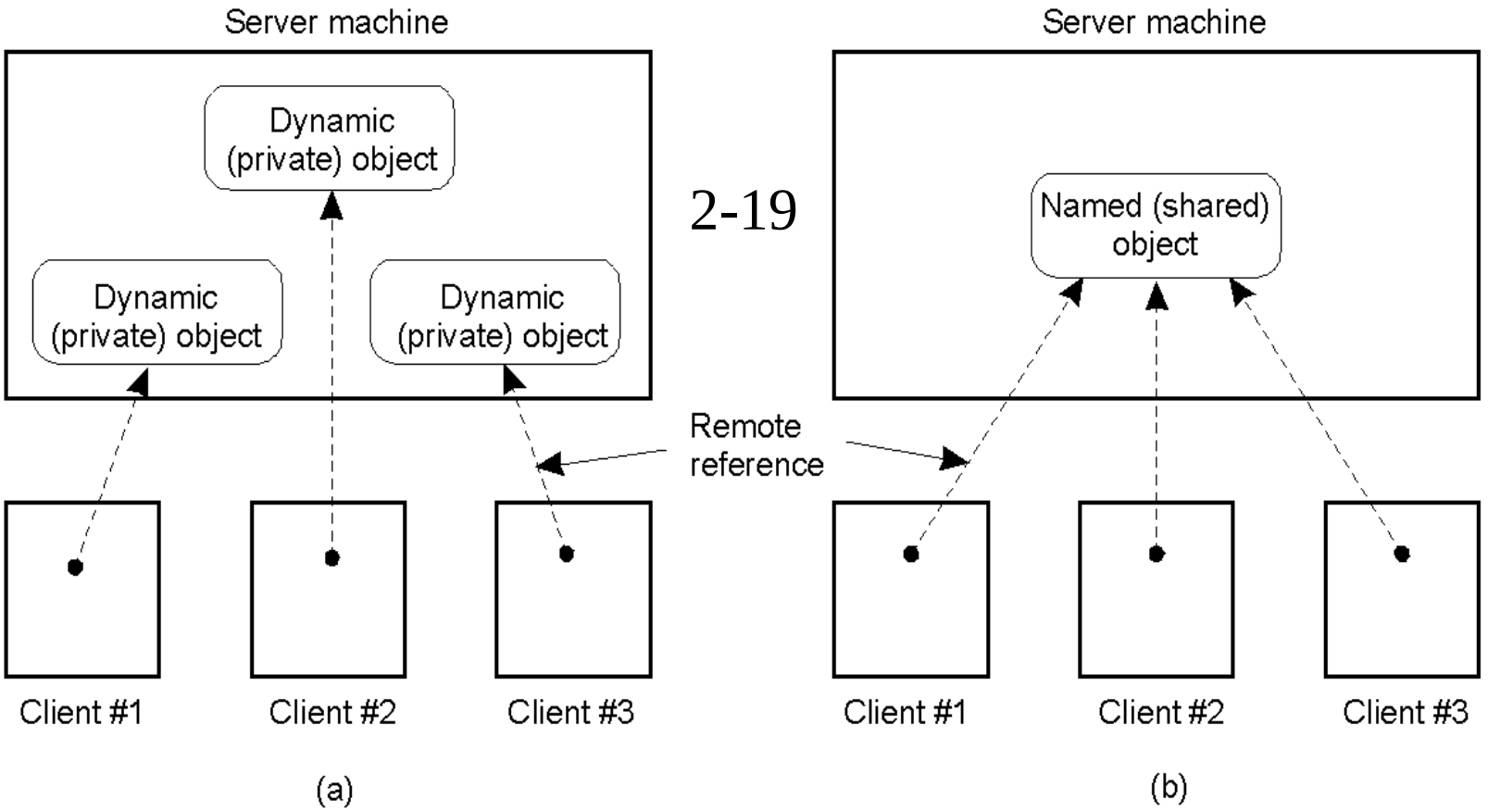
DCE's RPC mechanism has been "enhanced" to directly support remote method invocation.

- DCE Objects = xIDL plus C++.
- DCE IDL has been extended to support objects in C++.

Two types of DCE objects are supported:

- *Distributed Dynamic Objects* – a "private" object created by the server for the client.
- *Distributed Named Objects* – a "shared" object that lives on the server and can be accessed by more than one client.

The DCE Distributed-Object Model



a) DCE dynamic objects – requests for creation sent via RPC.

b) DCE named objects – registered with a DS naming service.

Example: Java RMI

- Distributed objects are integrated into the Java language.
  - high degree of *distribution transparency* (with exceptions).
  - Java supports distributed objects (not RPC).
- Distributed object's state is stored on the server, with interfaces made available to remote clients (via distributed object proxies).
- Programmer simply implements
  - client proxy as a class
  - server skeleton as another class.

Message-Oriented Middleware: MOM

As a communications mechanism, RPC/RMI is often inappropriate:

- what happens if the receiving side is “not awake” and waiting to communicate?
- default “synchronous (blocking” nature of RPC/RMI) is often *too restrictive*.

Something else is needed: **Messaging**.

xSome DS Comms. Terminology

- *Persistent Communications:*
  - Once sent, the "sender" can stop executing.
  - The "receiver" need not be operational at this time – the commmunications system **buffers** the message as required (until it can be delivered).

  [Can you think of an example?]

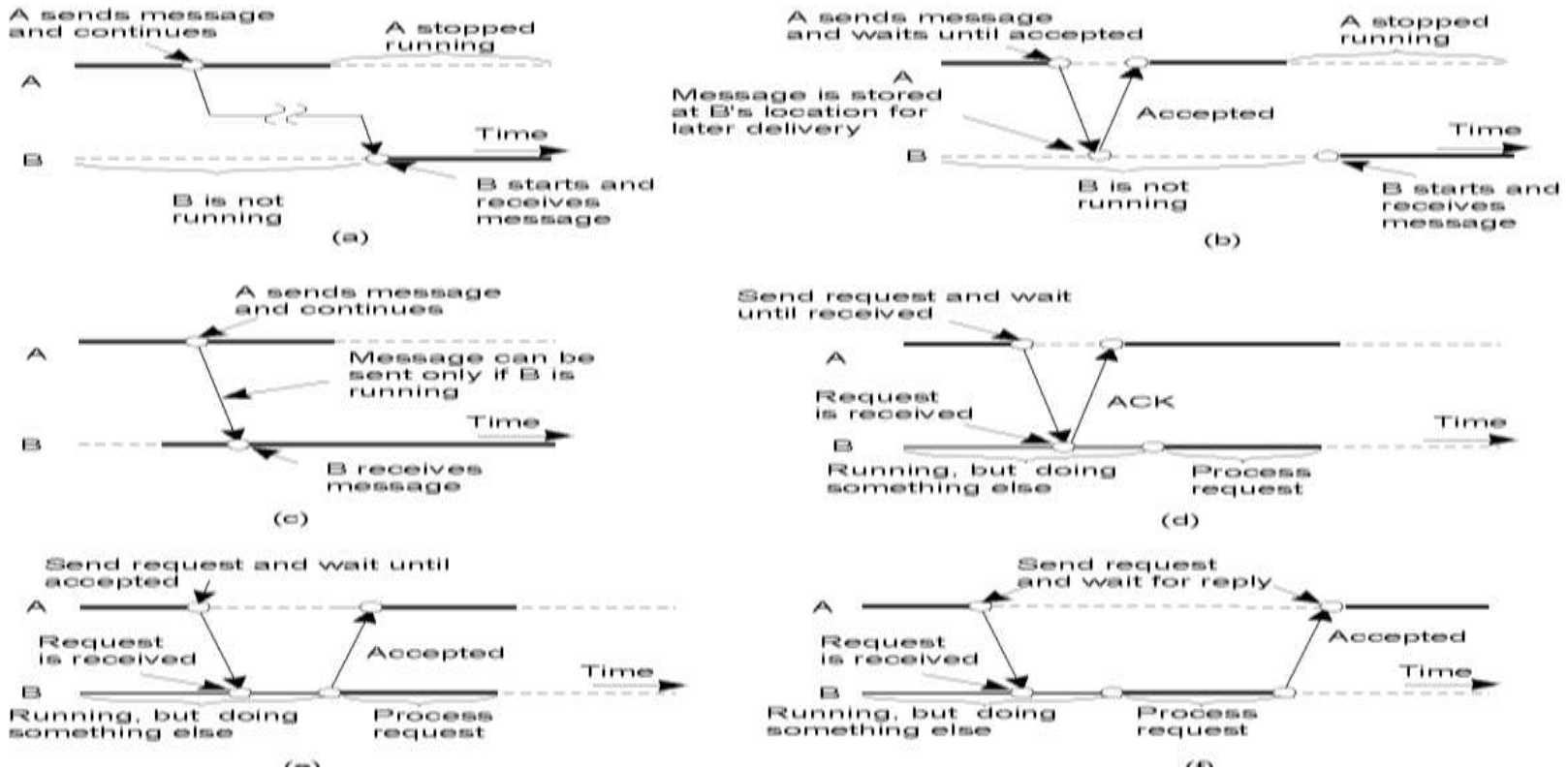
- *Transient Communications:*
  - The message is only stored as long as the "sender" and "receiver" are executing.
  - If problems occur, the message is **discarded**…

xMore DS Comms. Terminology

- *Asynchronous Communications:*
  - Sender **continues** with other work immediately upon sending a message to the receiver.

- *Synchronous Communications:*
  - Sender **blocks**, **waiting** for a reply from the receiver before doing any other work (default model for RPC/RMI technologies).

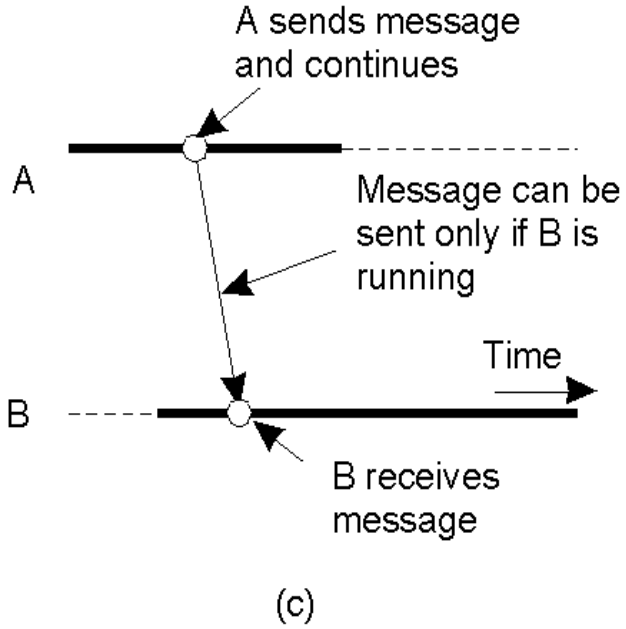
xClassifying Distributed Communications (1)



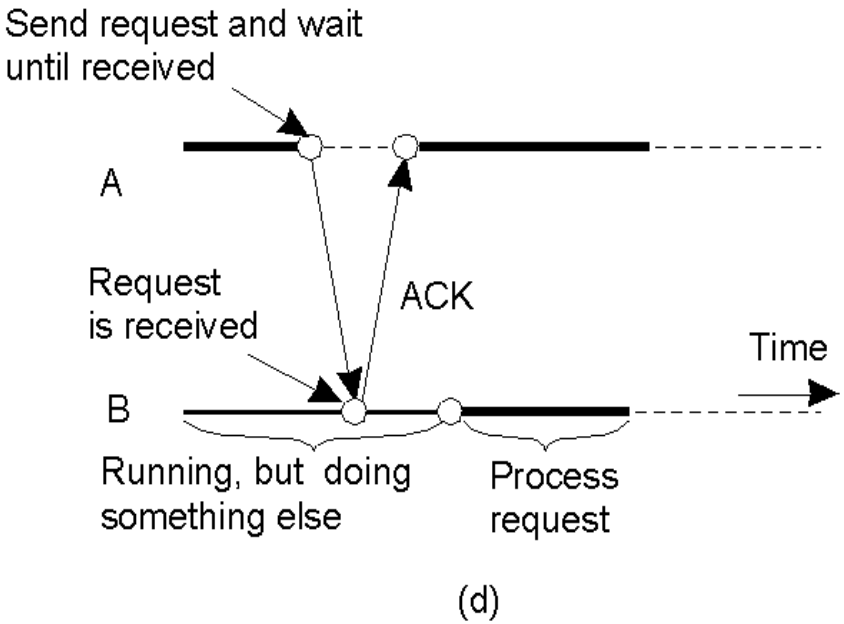
a) Persistent asynchronous communication.

b) Persistent synchronous communication.

xClassifying Distributed Communications (2)



(c)

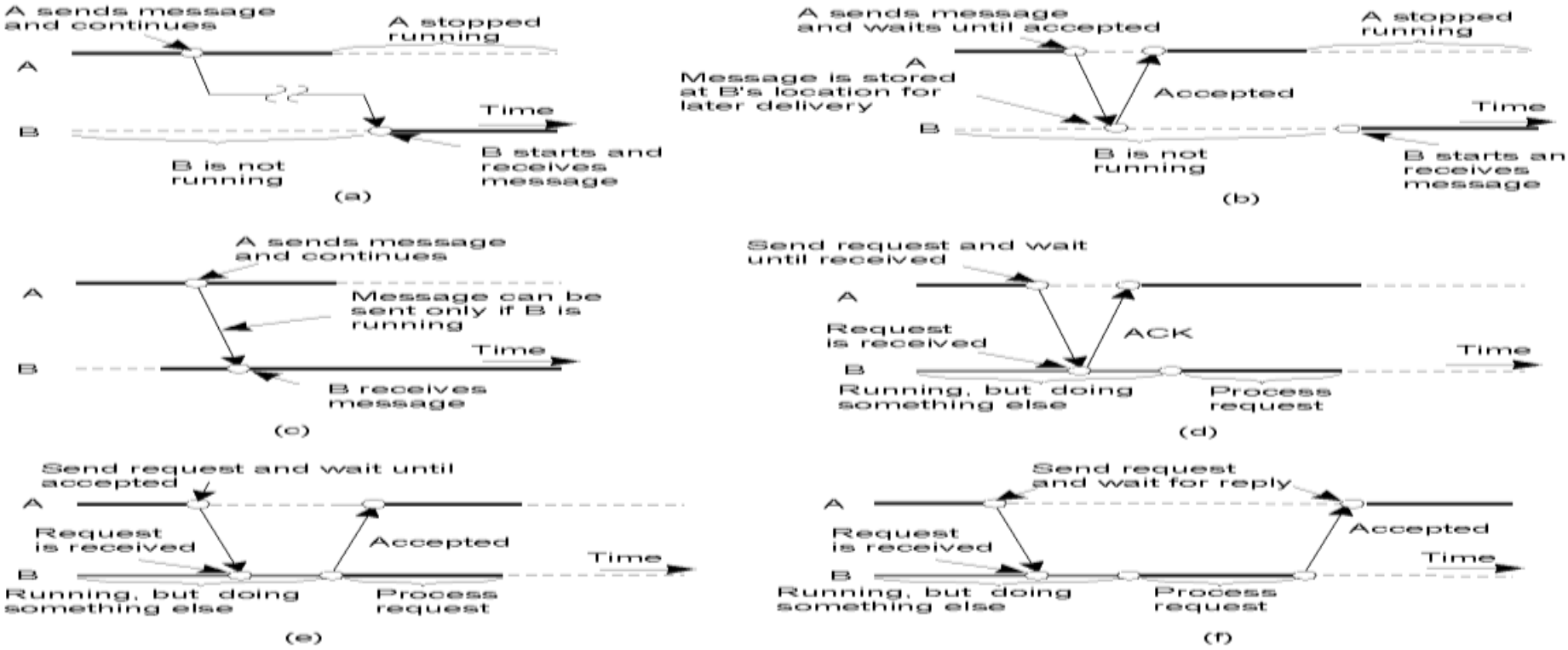


(d)

c) Transient asynchronous communication.

d) Receipt-based transient synchronous communication.

xClassifying Distributed Communications (3)



e) Delivery-based transient synchronous communication at message delivery.

f) Response-based transient synchronous communication.

Message Passing (MP) Systems

Fundamentally different approach.

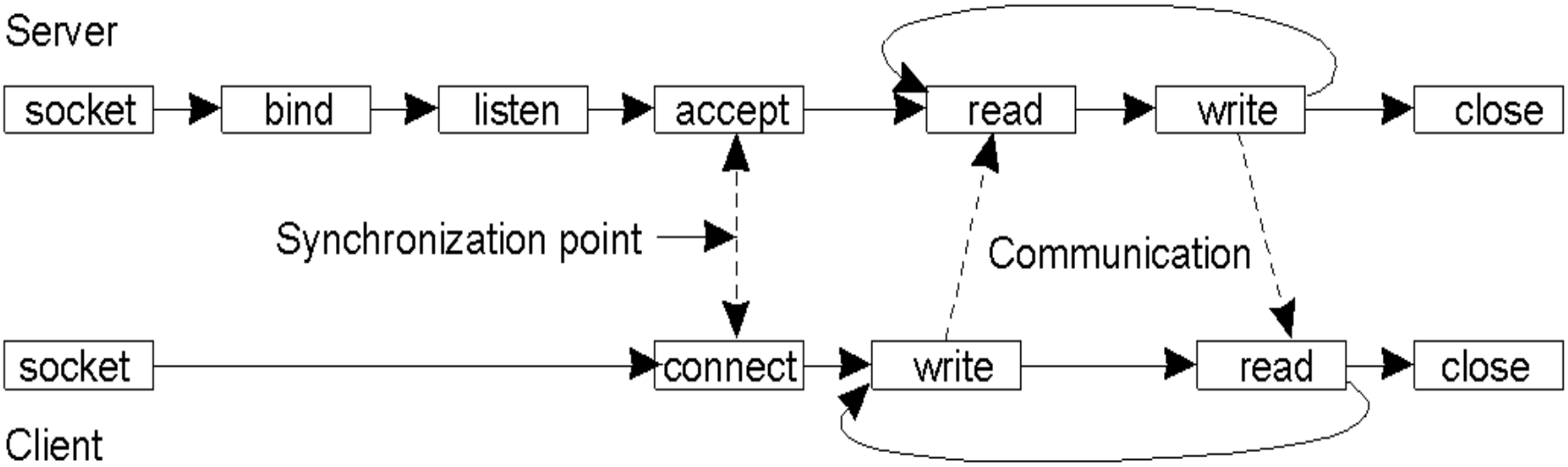
All communications primitives are defined in terms of passing “messages”.

Initially, MP systems were “transient”, but these did not scale well *geographically*.

Recent emphasis has been on “persistent” solutions.

Message-Oriented Transient Comms.

Initital efforts relied on the Sockets API.



However, DS developers rejected Sockets:

- Wrong level of abstraction (only “send” and “receive”).
- Too closely coupled to TCP/IP networks – not diverse enough.

The Message-Passing Interface (MPI)

Middleware vendors looked to provide a higher-level of abstraction.

Every vendor did their own thing (which is typical).

As can be imagined, this lead to *portability problems*, as no too vendors product interfaces were the same.

The solution?

The “Message-Passing Interface” (MPI).

The MPI API

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.

Some of the more intuitive (and useful) message-passing primitives (Note: there are many more in the API).

Message-Oriented Persistent Comms.

Also known as: “message-queuing systems”.

They support persistent, asynchronous communications.

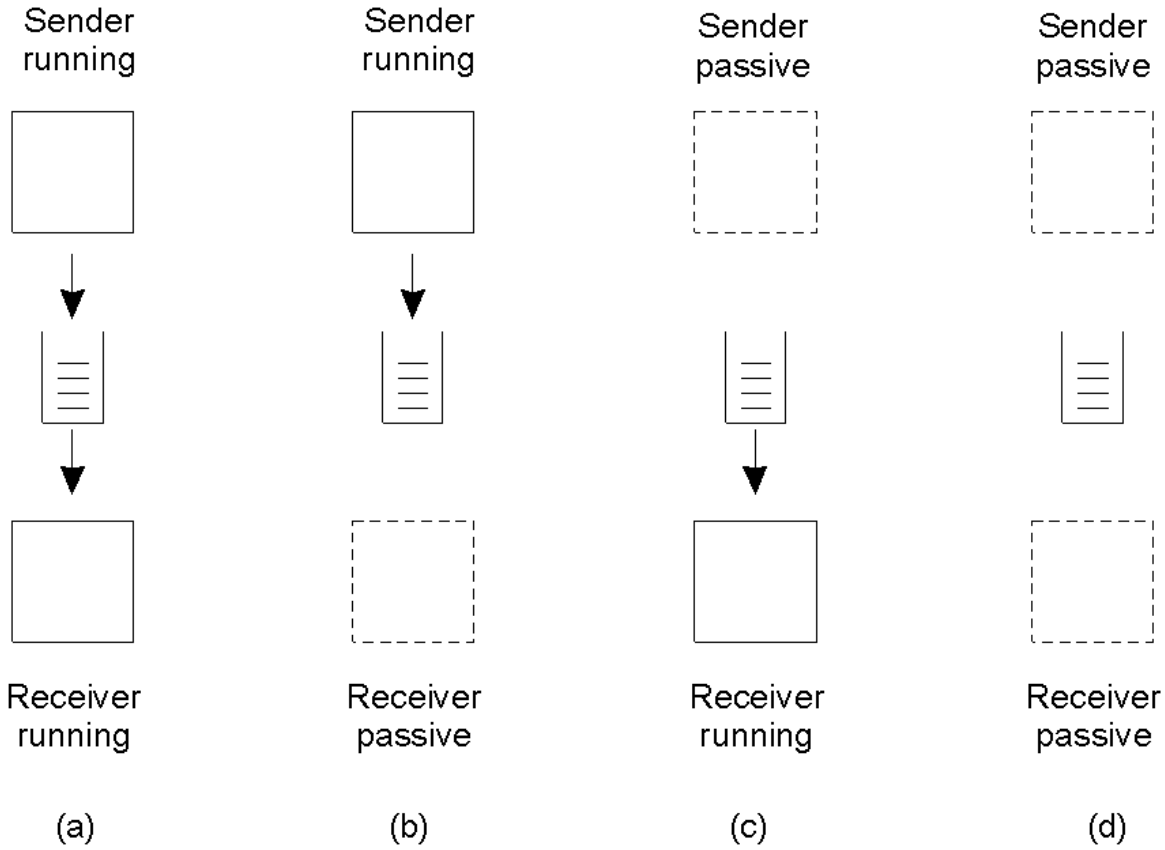
Typically, transport can take minutes (hours?) as opposed to seconds/milliseconds.

The basic idea: applications communicate by putting messages into and taking messages out of “message queues”.

Only guarantee: your message will eventually make it into the receiver’s message queue.

This leads to “loosely-coupled” communications.

Message-Queuing Models



Four combinations for “loosely-coupled” communications which use message-queues.

Message-Queuing API

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.

Basic interface to a queue in a message-queuing system: this is a very simple, yet *extremely powerful* abstraction.

Message-Queuing System Architecture

Messages are “put into” a *source queue*.

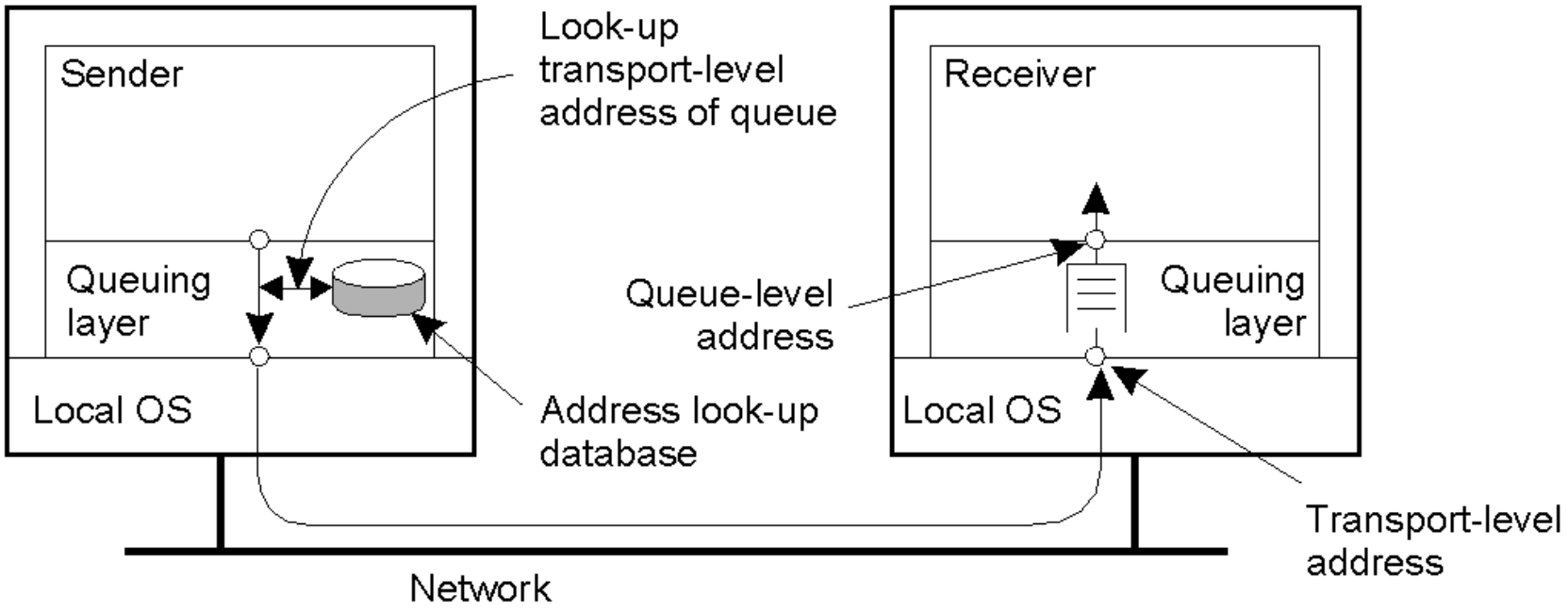
They are then “taken from” a *destination queue*.

Obviously, a mechanism has to exist to move a message from a source queue to a destination queue.

This is the role of the *Queue Manager*.

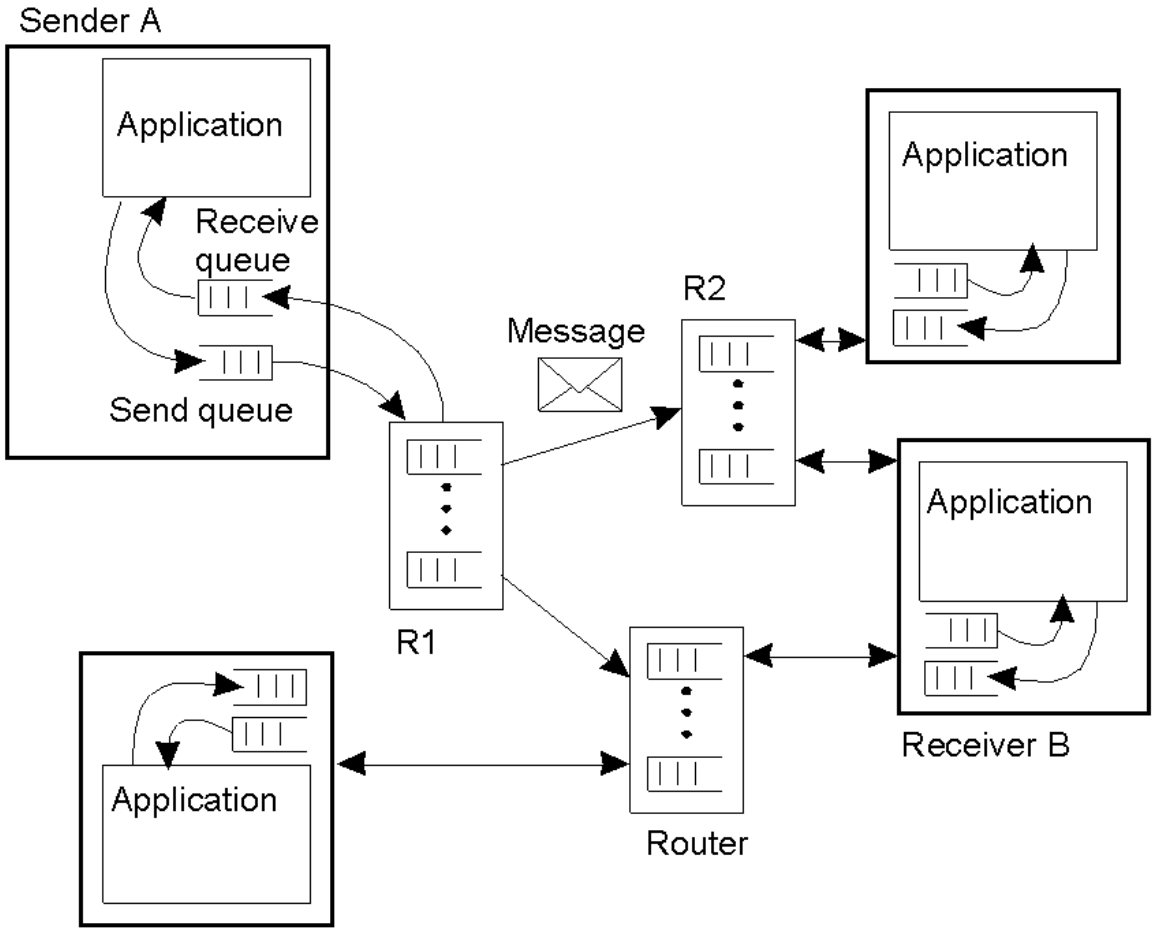
These are message-queuing “relays” that interact with the distributed applications and with each other. Not unlike routers, these devices support the notion of a DS “overlay network”.

General Architecture of a Message-Queuing System (1)



The relationship between queue-level addressing and network-level addressing. The queuing layer is at a higher level of abstraction that the underlying network.

General Architecture of a Message-Queuing System (2)



The general organization of a message-queuing system with routers. The Queue Managers can reside within routers as well as within the DS end-systems.

The Role of Message Brokers

Often, there's a need to integrate new/existing apps into a "single, coherent Distributed Information System (DIS)".

In other words, it is not always possible to start with a *blank page* – distributed systems have to live in the real world.

Problem: different message formats exist in legacy systems (cooperation and adherence to open standards was not how things were done in the past).

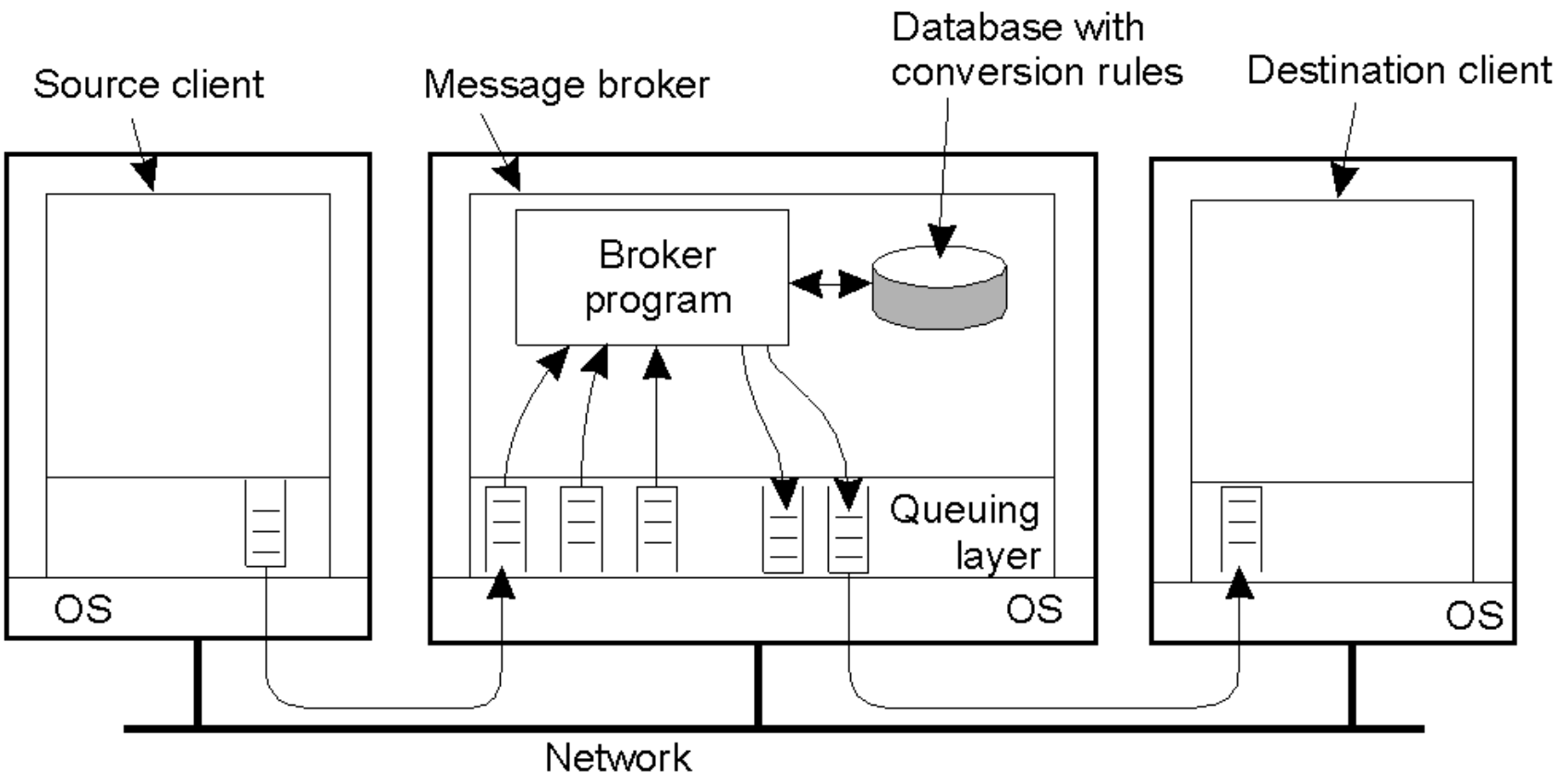
It may not be convenient to "force" legacy systems to adhere to a single, global message format (cost!?).

It is often necessary to live with diversity (there's no choice).

How?

Meet the "Message Broker".

Message Broker Organization



The general organization of a message broker in a message-queuing system – also known variously as an “interface engine”.

Message-Queuing (MQ) Applications

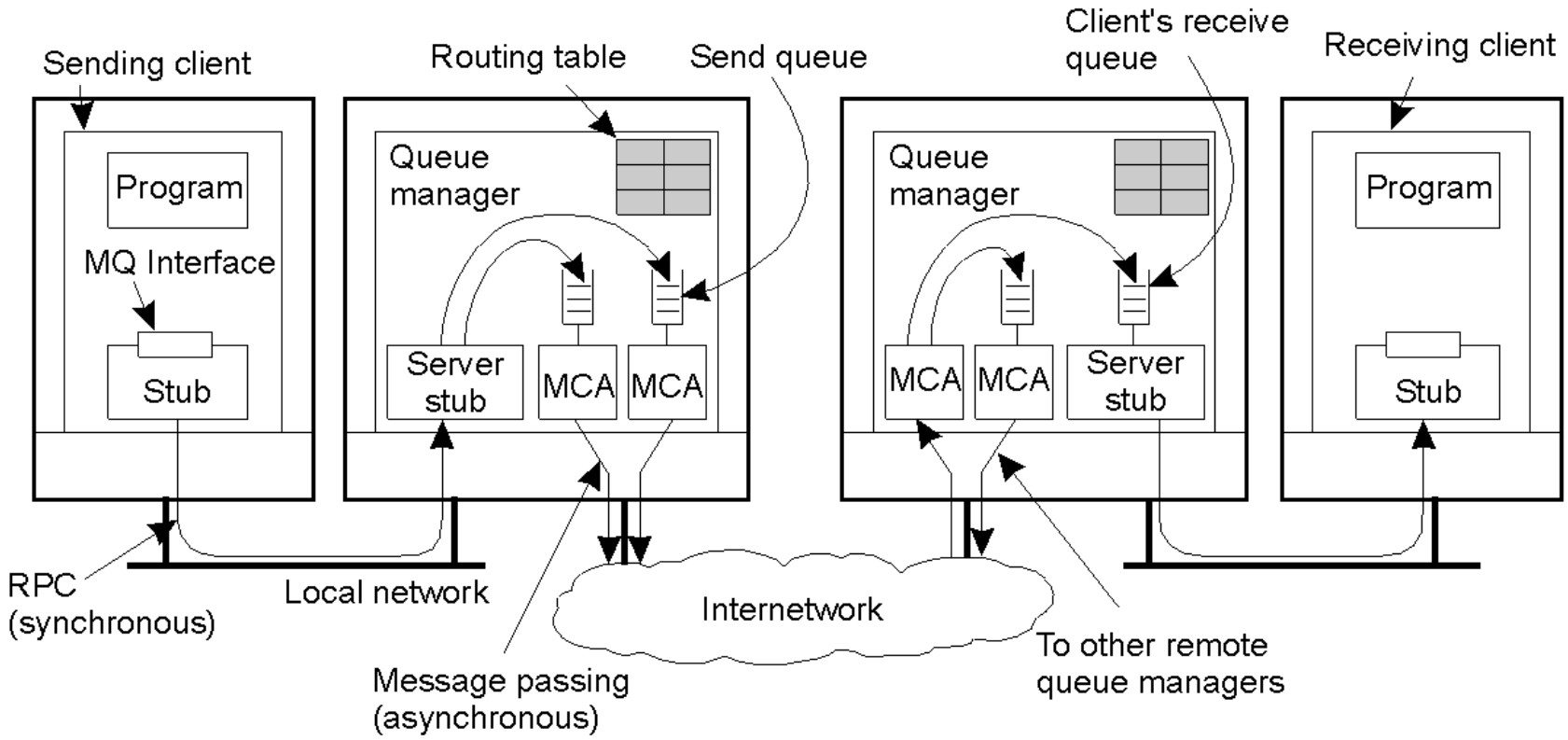
General-purpose MQ systems support a wide range of applications, including:

- Electronic mail.
- Workflow.
- Groupware.
- Batch Processing.

Most important MQ application area:

The integration of a widely dispersed collection of **database applications** (which is all but impossible to do with traditional RPC/RMI techniques).

Example: IBM MQSeries



General organization of IBM's MQSeries message-queuing system.

IBM MQSeries: Message 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.

Some attributes associated with message channel agents (MCA).

IBM MQSeries: Message Transfer

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.

Stream-Oriented Communications

With RPC, RMI and MOM, the effect that time has on correctness is of *little consequence*.

However, audio and video are *time-dependent data streams* – if the timing is off, the resulting "output" from the system will be incorrect.

Time-dependent information – known as "continuous media" communications.

Example: voice: PCM: 1/44100 sec intervals on playback.

Example: video: 30 frames per second (30-40 msec per image).

KEY MESSAGE: Timing is crucial!

Transmission Modes

- ***Asynchronous** transmission mode*
  - data stream is transmitted in order, but there's no timing constraints placed on the actual delivery (e.g., File Transfer).
- ***Synchronous** transmission mode*
  - maximum end-to-end delay is defined (but data can travel faster).
- ***Isochronous** transmission mode*
  - data transferred "on time" – there's a **maximum** and **minimum** end-to-end delay (known as "bounded jitter").
  - Known as "streams" – isochronous transmission mode is very useful for multimedia systems.

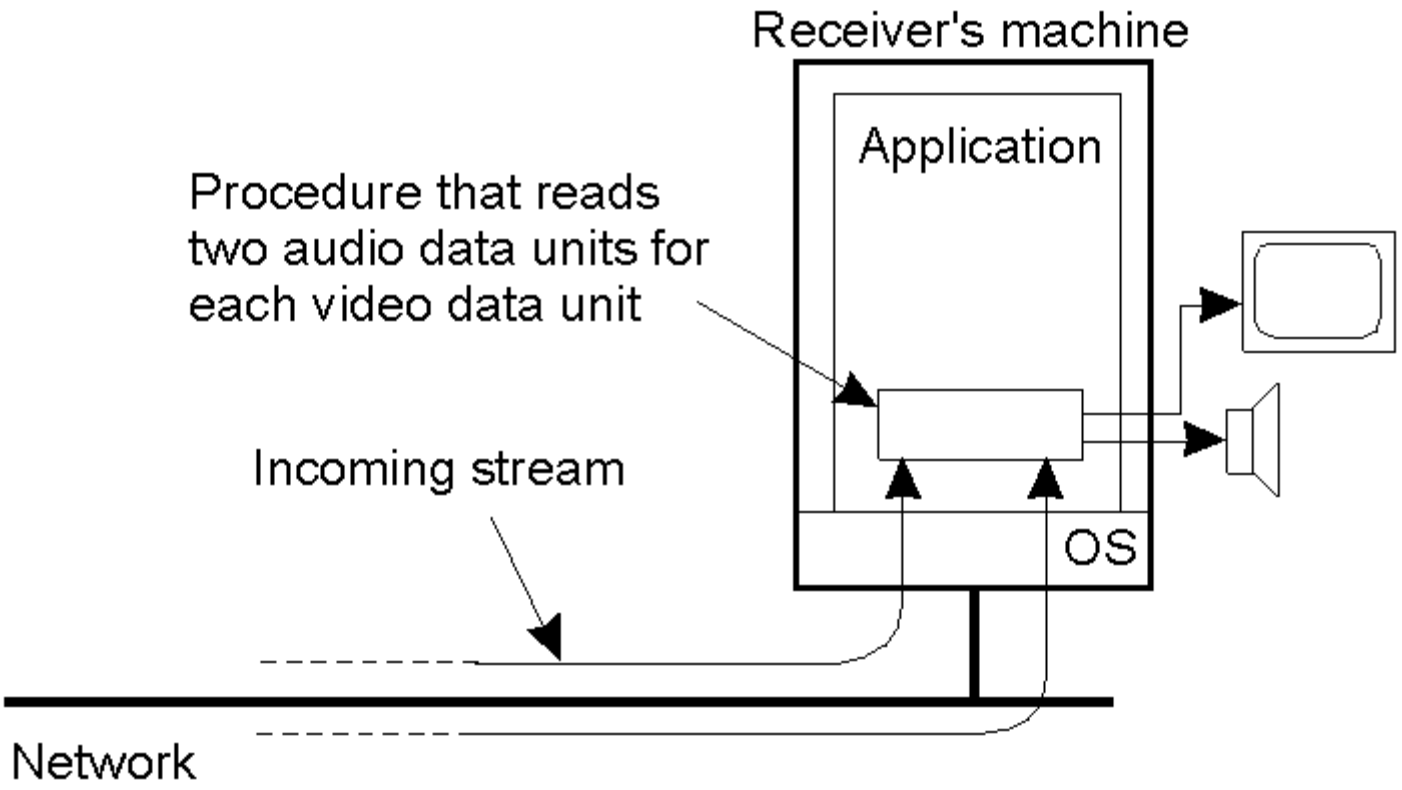
Two Types of Streams

- ***Simple** Streams*
  - one single sequence of data, for example: voice.

- ***Complex** Streams*
  - several sequences of data (sub-streams) that are "related" by time.
    - e.g. lip-synchronized movie (sound and pictures, together with sub-titles)…

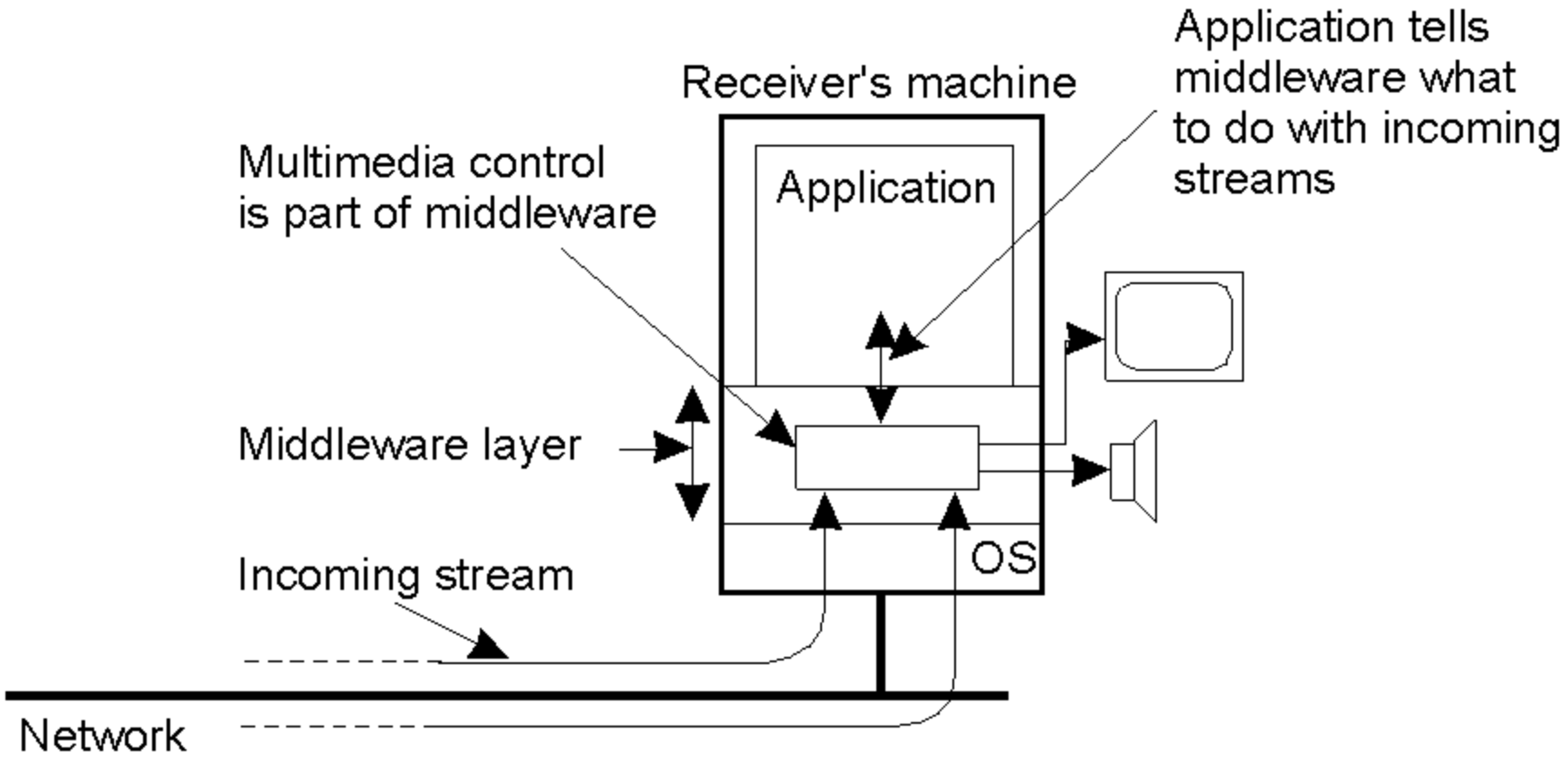
- This leads to data synchronization problems… which are not at all easy to deal with.

Explicit Synchronization



The principle of explicit synchronization on the level data units for multiple streams (sub-streams).

Higher-Level Synchronization



The principle of synchronization as supported by high-level interfaces built as a set of “multimedia middleware streaming services”.

Synchronization

A key question is:

"Where does the synchronization occur?"

On the sending side?

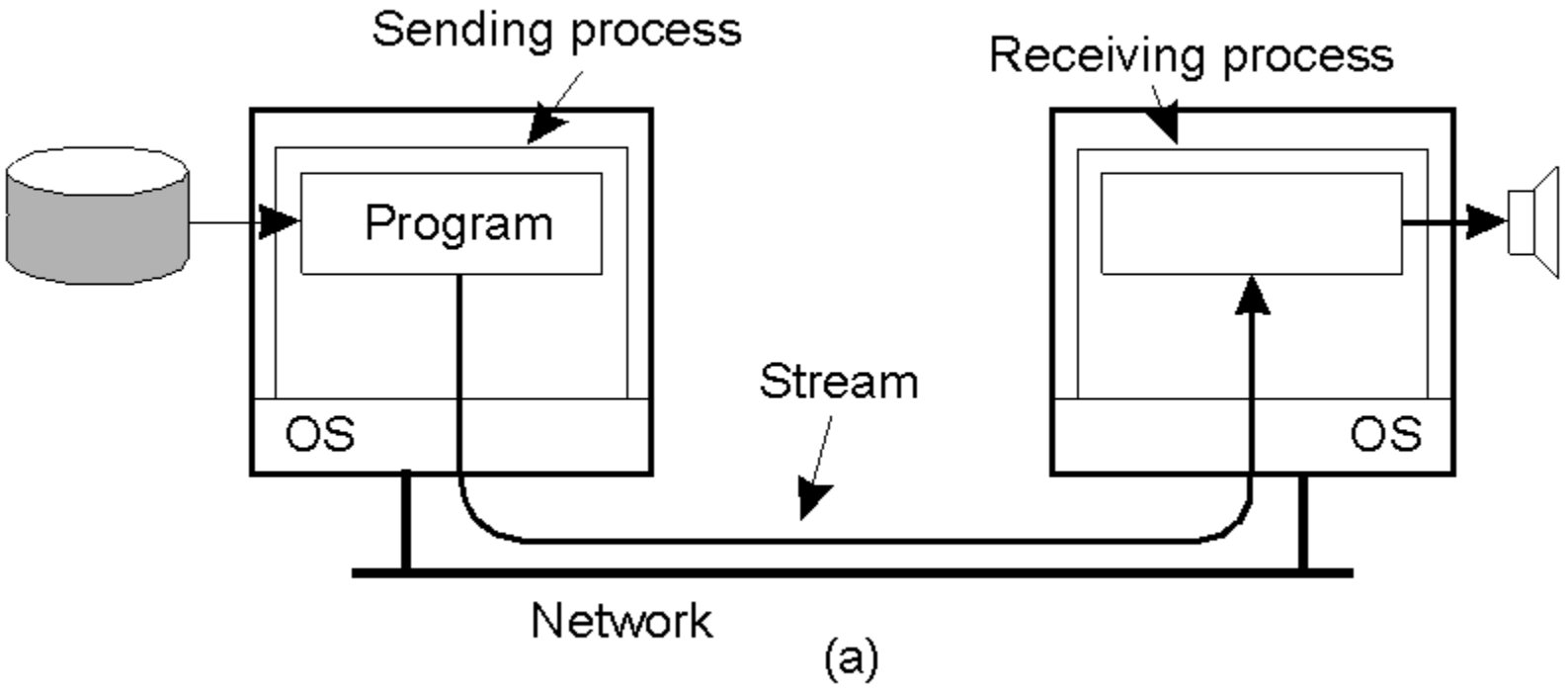
On the receiving side?

Think about the advantages/disadvantages of each …

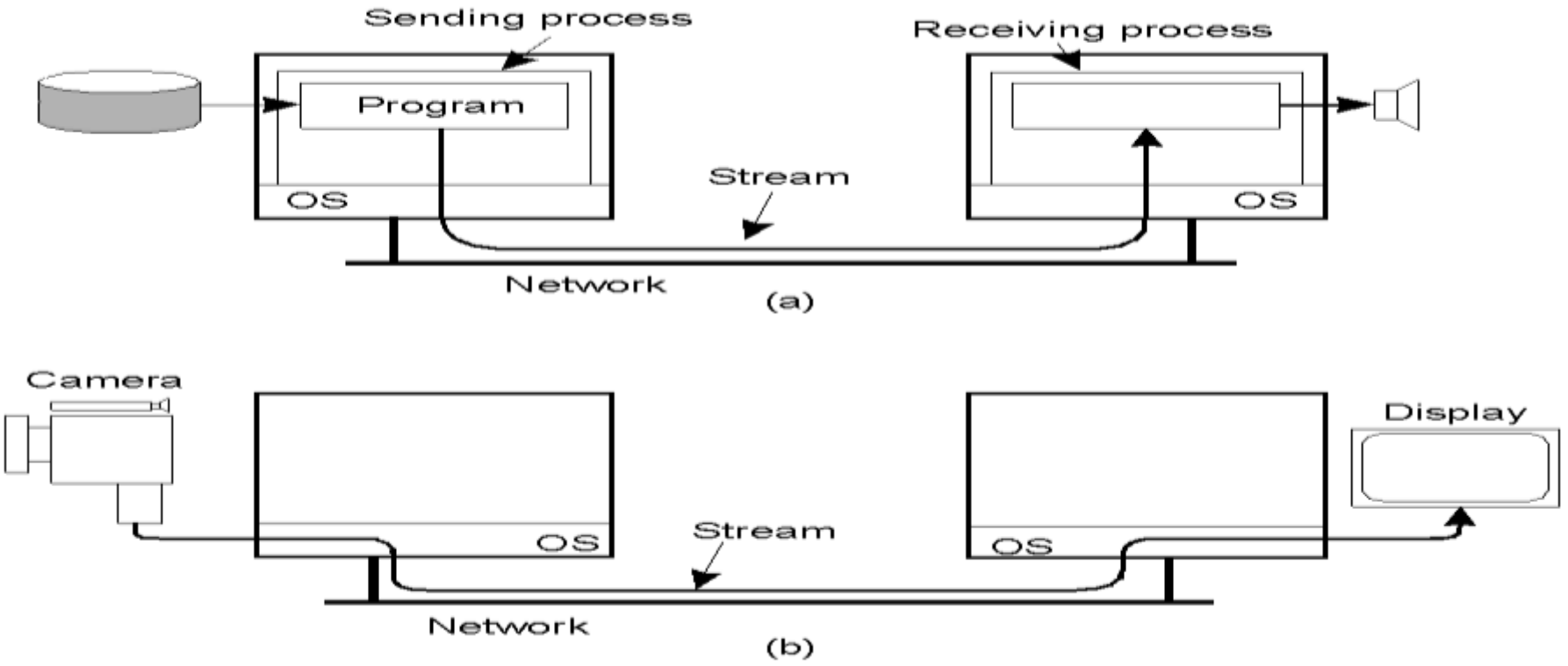
Components of a Stream

Two parts: a “source” and a “sink”.

The source and/or the sink may be a networked process (a) or an actual end-device (b).

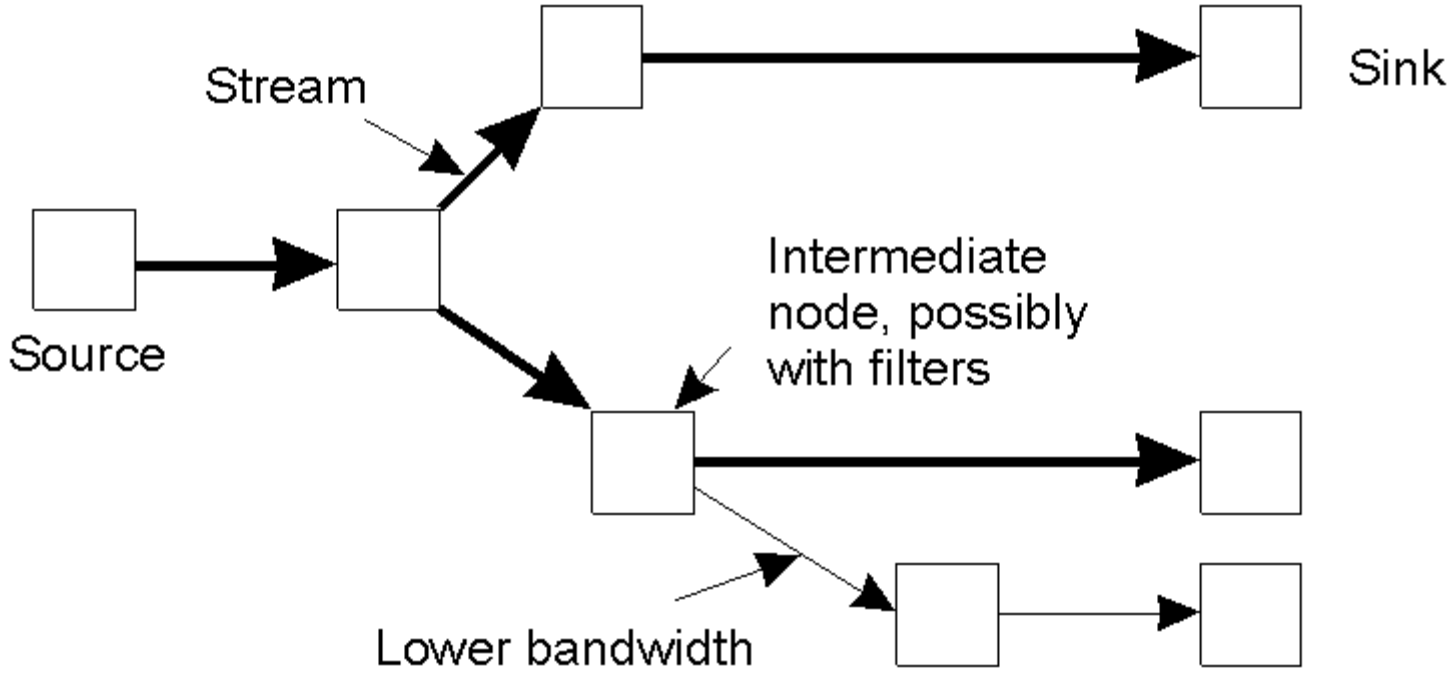


End-device to End-device Streams



Setting up a stream directly between two devices – i.e., no inter-networked processes.

Multi-party Data Streams



An example of multicasting a stream to several receivers. This is "multiparty communications" – different delivery transfer rates may be required by different end-devices.

Quality of Service (QoS)

- Definition: “ensuring that the temporal relationships in the stream can be preserved”.

- QoS is all about three things:
  - (a) Timeliness, (b) Volume and (c) Reliability.

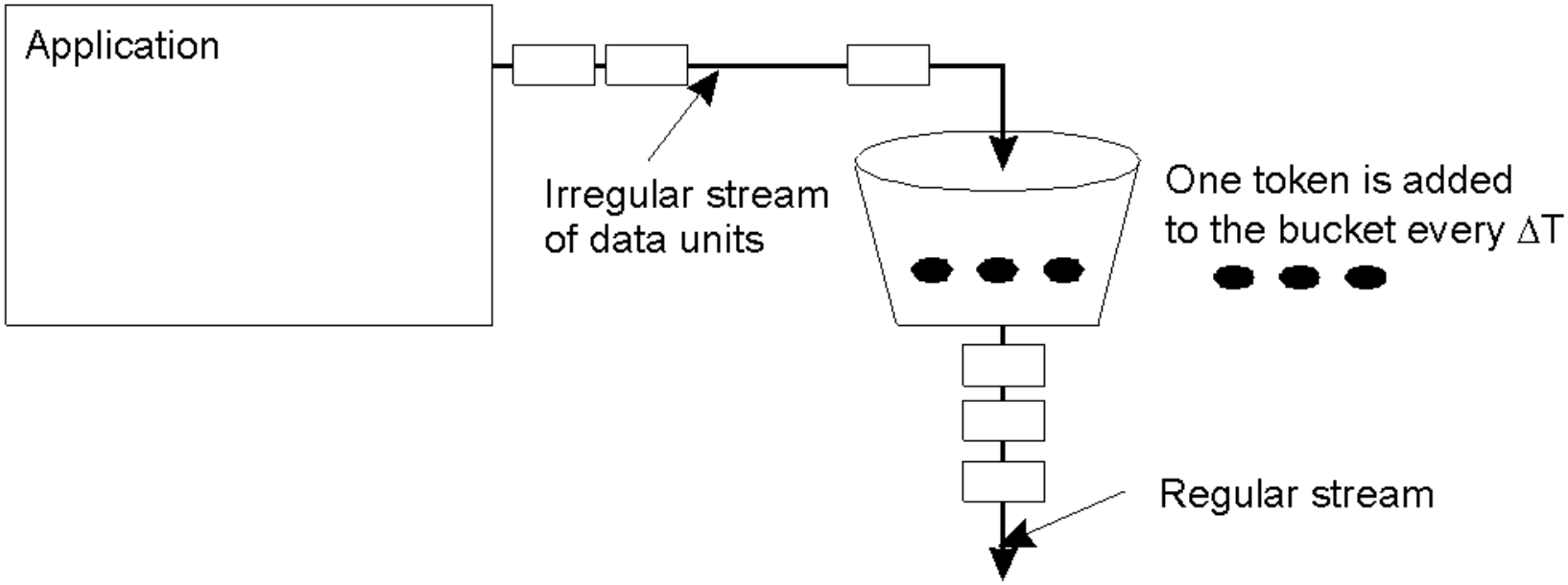
- But, how is QoS actually specified?
  - Unfortunately, most technologies do their own thing.

Specifying QoS with Flow Specs.

Characteristics of the Input	Service Required
•maximum data unit size (bytes) •Token bucket rate (bytes/sec) •Toke bucket size (bytes) •Maximum transmission rate (bytes/sec)	•Loss sensitivity (bytes) •Loss interval (μsec) •Burst loss sensitivity (data units) •Minimum delay noticed (μsec) •Maximum delay variation (μsec) •Quality of guarantee

A flow specification – one way of specifying QoS – a little complex, but it does work (but not via a user controlled interface).

An Approach to Implementing QoS



The principle of a token bucket algorithm – a "classic" technique for controlling the flow of data (and implementing QoS characteristics).

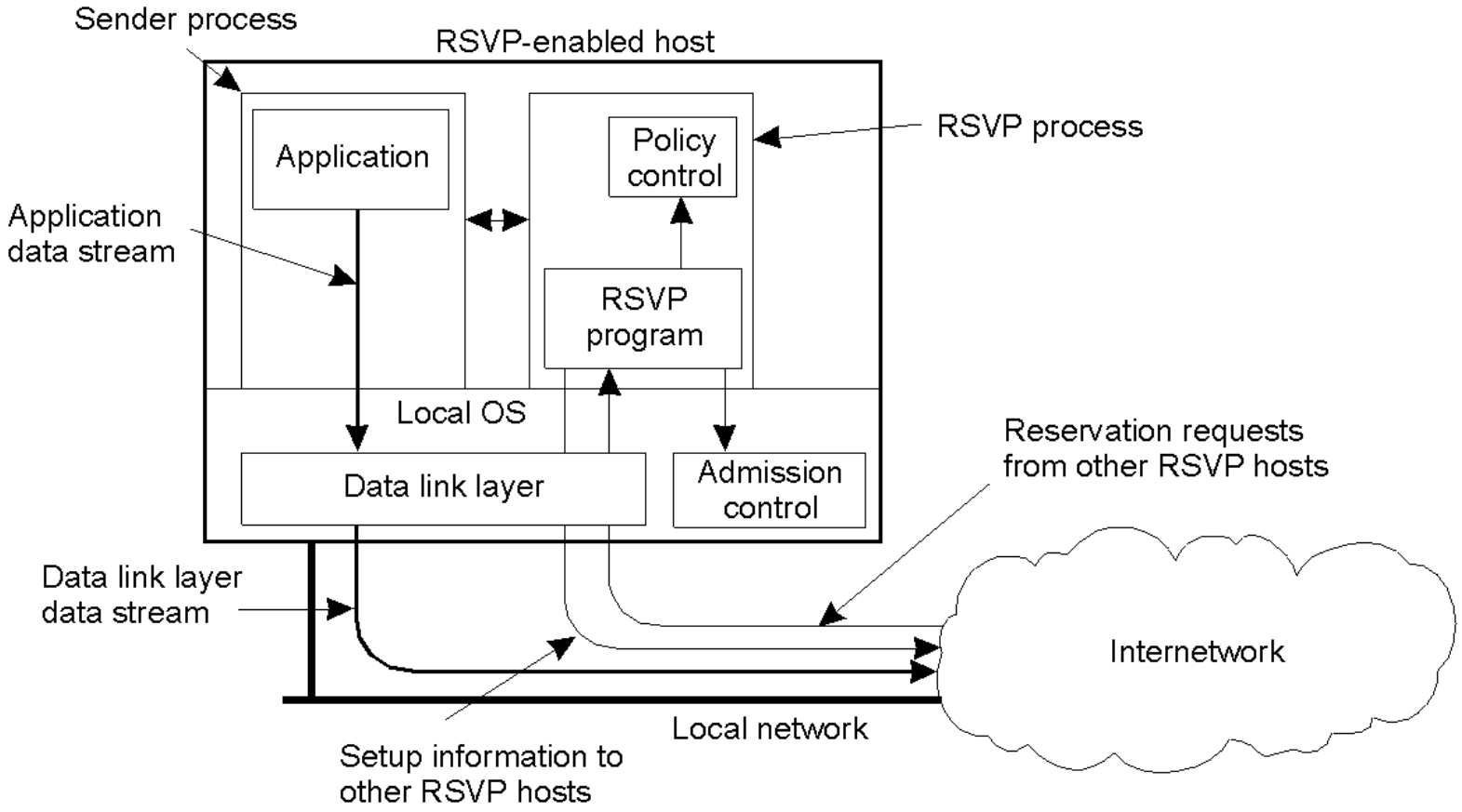
Stream Management

- Managing streams is all about managing bandwidth, buffers, processing capacity and scheduling priorities – which are all needed in order to realise QoS guarantees.

- This is not as simple as it sounds, and there's no general agreement as to "how" it should be done.

  - ATM's QoS (which is very "rich") has proven to be unworkable (difficult to implement).
  - Another technique is the Internet's RSVP.

Internet RSVP QoS



The basic organization of RSVP for resource reservation in a distributed system – transport-level control protocol for enabling resource reservations in network routers. Interesting characteristic: *receiver initiated*.

Distributed Comms. - Summary

- Power and flexibility are essential, as network programming primitives are too “primitive”.

- Middleware Comms. Mechanisms – providing support for a higher-level of abstraction.

  - RPC and RMI: synchronized, transient.
  - MOM: convenient, asynchronous, persistent.
  - Streams: deal with “temporally related data” (not easy).