Chapter 4

Communication

October, 8th

Introduction

- Interprocess communication is fundamental
- Building a system based on low-level message passing (i.e., as offered by the hardware) is difficult and not secure
- Higher-level abstractions and protocols can help
- Models of communication
  - Remote Procedure Call (RPC)
  - Message-Oriented Middleware (MOM)
  - Multicast communication

Fundamentals

Layered protocols

- Communication principle
  - Process A can communicate with process B by building a message in its address space and making a system call to send it to B
- In reality, many agreements are needed for communication
  - How many volts should be used to signal a 0-bit and a 1-bit?
  - How does the receiver know the last bit of the message?
  - How long are numbers, strings, and other data items?
  - How to detect if a message has been damaged or lost?

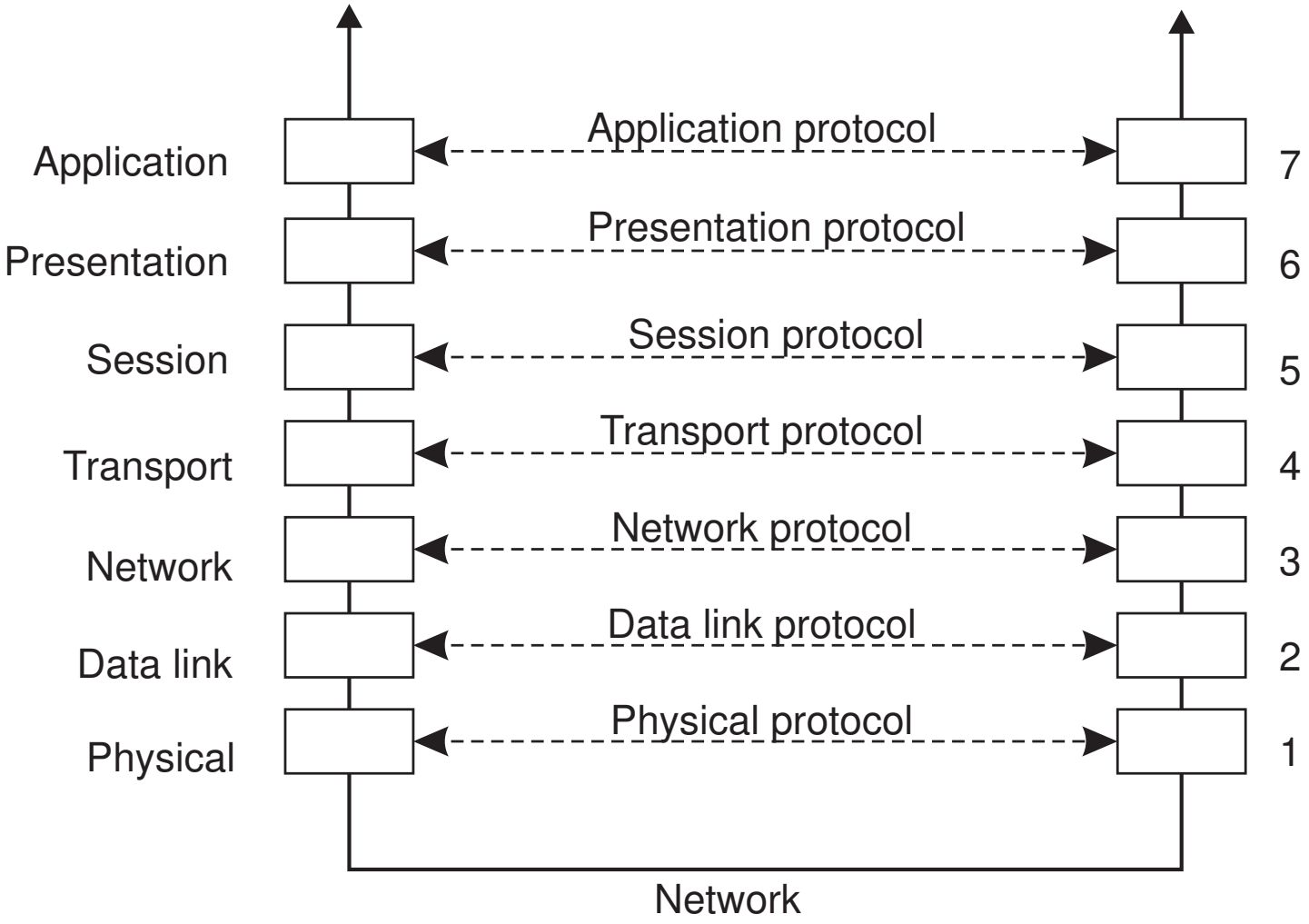
Layered Protocols

OSI Model

- Defined by the International Standards Organization (ISO)
- Never widely used and implemented
- **Why?** allow open systems to communicate
- **How?** with layered protocols, providing communication services
  - connection-oriented services
  - connectionless services

Layered Protocols

OSI Model

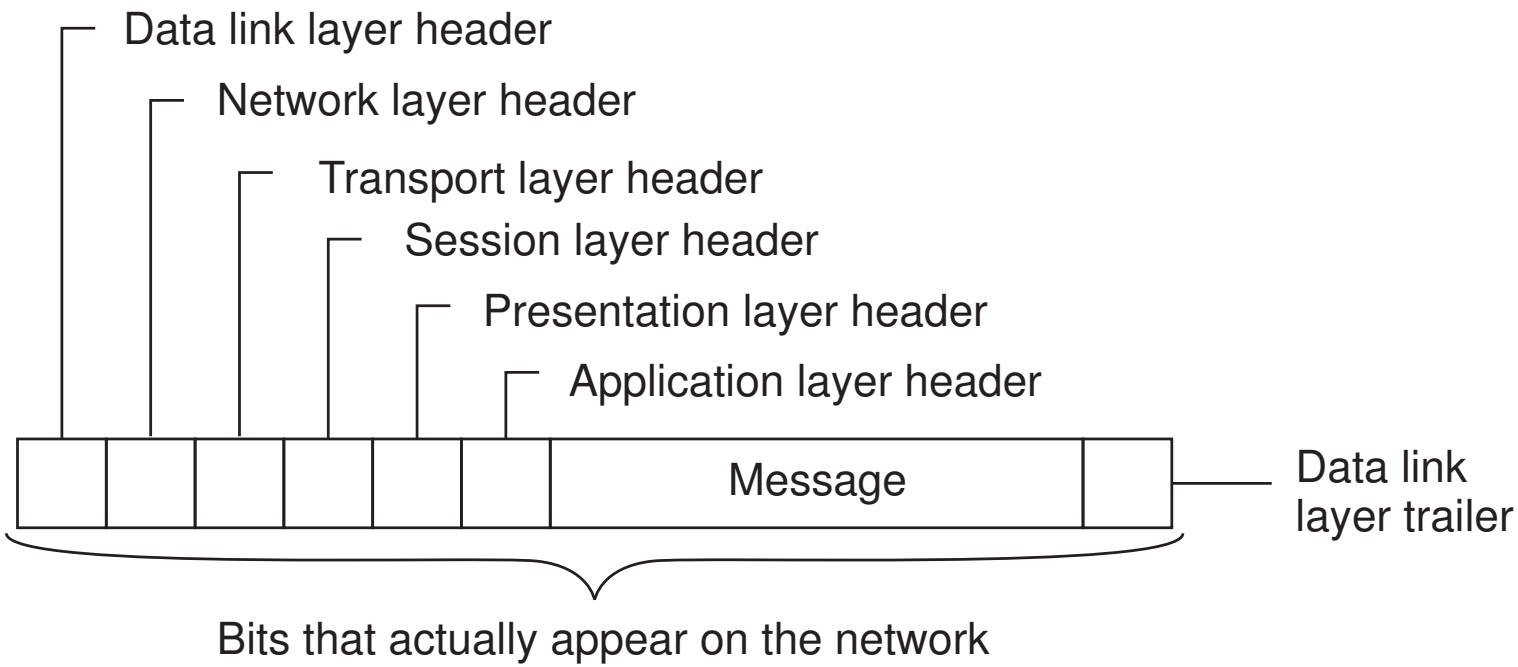


Layers, interfaces, and protocols in the OSI model

Layered Protocols

OSI Model

1. Process A builds message and passes it to application layer (e.g., usually library function)
2. Application layer adds header to message and passes it to the presentation layer,
3. ...



A typical message as it appears on the network

Lower-level protocols

Three lowest layers of the OSI protocol suite:

- Network layer
- Data link layer
- Physical layer

Lower-level protocols

Three lowest layers of the OSI protocol suite:

- Network layer
- Data link layer
- Physical layer
  - how to **send bits** from one end to the other (e.g., RS-232-C)
  - deals with electrical, mechanical and signaling interfaces
  - does not handle errors

Lower-level protocols

Three lowest layers of the OSI protocol suite:

- Network layer
- Data link layer
  - groups bits into units (frames) and **ensures correct transmission**
  - marks start and end of the with a special bit pattern
  - computes and appends a checksum to frame
  - receiver uses checksum to check correctness of received frame
- Physical layer
  - how to **send bits** from one end to the other (e.g., RS-232-C)
  - deals with electrical, mechanical and signaling interfaces
  - does not handle errors

Lower-level protocols

Three lowest layers of the OSI protocol suite:

- Network layer
  - concerned with **routing** messages from origin to destination
  - multiple routes typically exist: how to choose the best one?
- Data link layer
  - groups bits into units (frames) and **ensures correct transmission**
  - marks start and end of the with a special bit pattern
  - computes and appends a checksum to frame
  - receiver uses checksum to check correctness of received frame
- Physical layer
  - how to **send bits** from one end to the other (e.g., RS-232-C)
  - deals with electrical, mechanical and signaling interfaces
  - does not handle errors

Transport protocols

Provide the minimal network protocol stack

- Transmission Control Protocol (TCP)
  - reliable communication
  - connection-oriented
- Universal Datagram Protocol (UDP)
  - unreliable (but more efficient) communication
  - connectionless

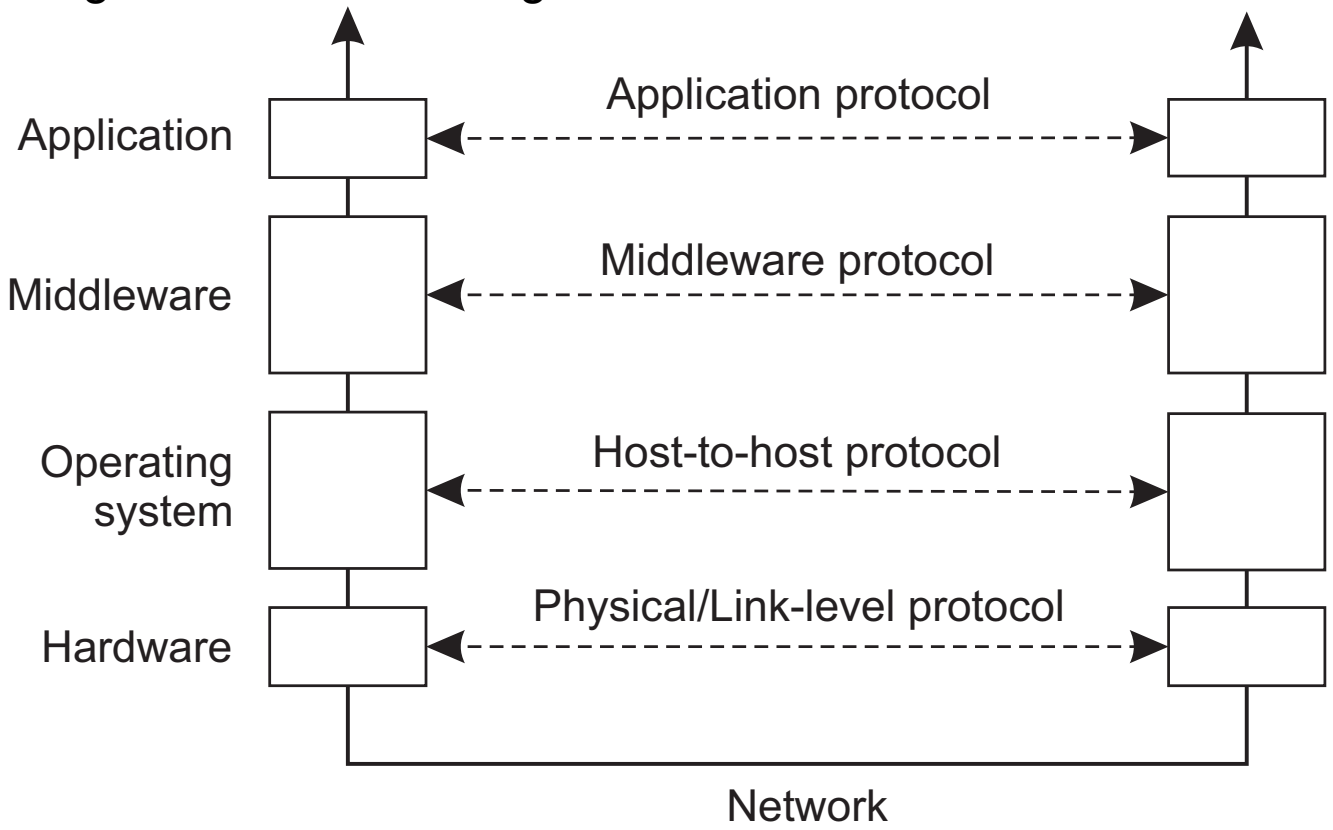
Higher-level protocols

Usually grouped together with the application

- Session layer
  - enhanced transport layer (e.g., dialog control, checkpointing)
  - rarely supported
- Presentation layer
  - simplifies communication between machines with different internal data representation
- Applications
  - everything else...
  - e.g., SMTP/IMAP (e-mail), FTP, HTTP

Middleware protocols

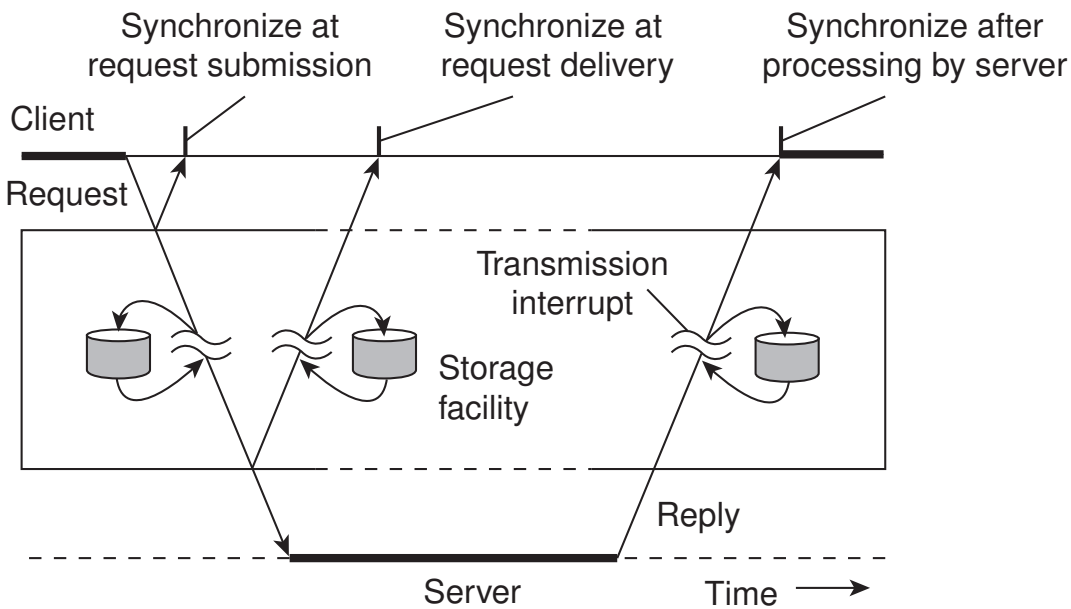
- Reside mostly at the application layer
- Application with general-purpose protocols with their own layers
  - e.g., authentication middleware (proof of a claimed identity)
  - e.g., middleware for commit protocols
  - e.g., distributed locking middleware



An adapted reference model for networked communication

Types of communication

- Persistent communication
  - message stored by the middleware until delivered to the receiver
- Transient communication
  - message is only delivered if both sender and receiver are executing
  - transport-layer communication services typically offer transient communication (e.g., store-and-forward routers)



Middleware as an intermediate service in application level communication.

Types of communication

- Asynchronous communication
  - sender continues immediately after submitting message
  - message is temporarily stored by the middleware
- Synchronous communication
  - sender is blocked until request is received; three options:
  - 1st: sender blocked until middleware takes over communication
  - 2nd: sender blocked until message delivered to recipient
  - 3rd: sender blocked until message processed by recipient

Remote Procedure Call

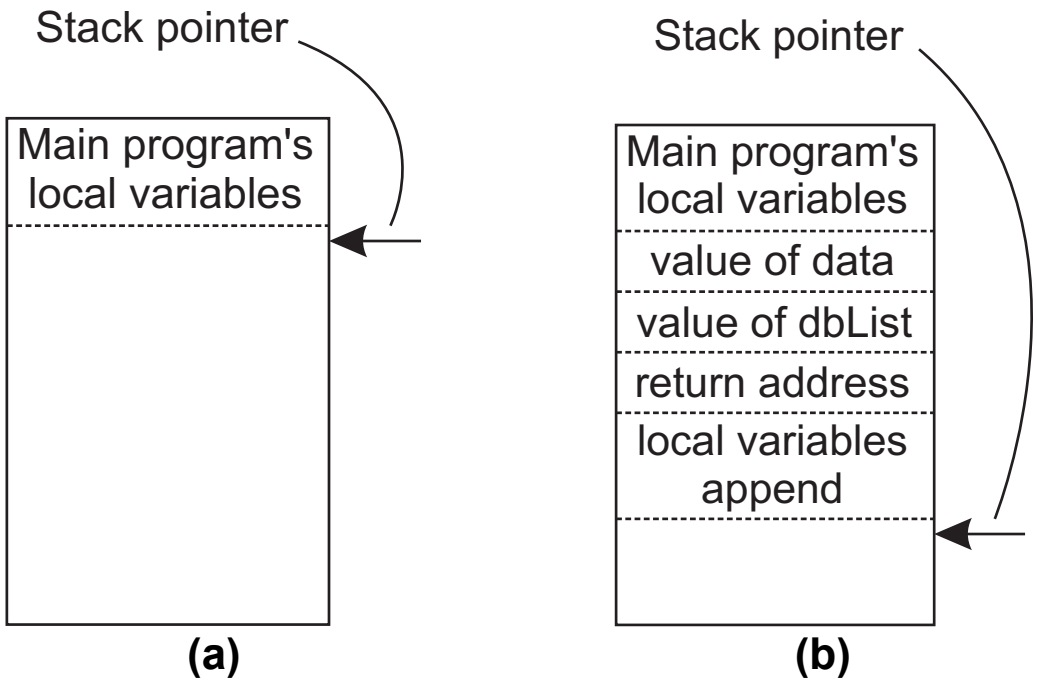
Introduction

- Simple communication mechanism
- More natural than send/receive primitives
- Programs call procedures located on other machines
- Implementation not trivial
  - Procedures in different address spaces
  - How to pass parameters across invocations and replies?
  - How to handle failures (of both caller and callee)?

Basic RPC operation

Conventional procedure call

Example: `newlist = append(data, dbList)`

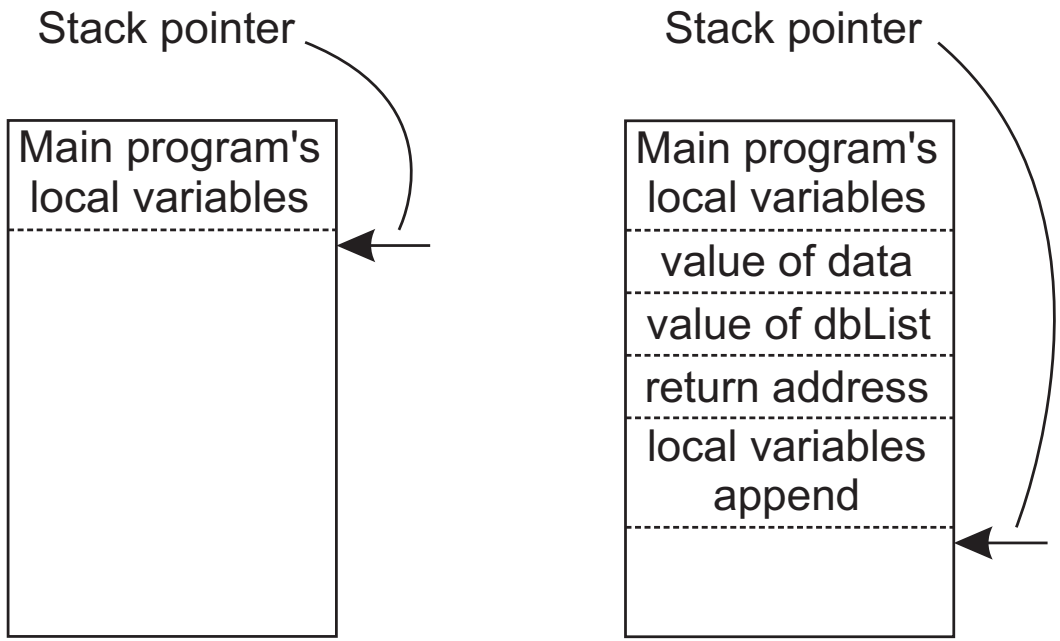


(a) Parameter passing in a local procedure call: the stack before the call to read. (b) The stack while the called procedure is active.

Basic RPC Operation

Call conventions

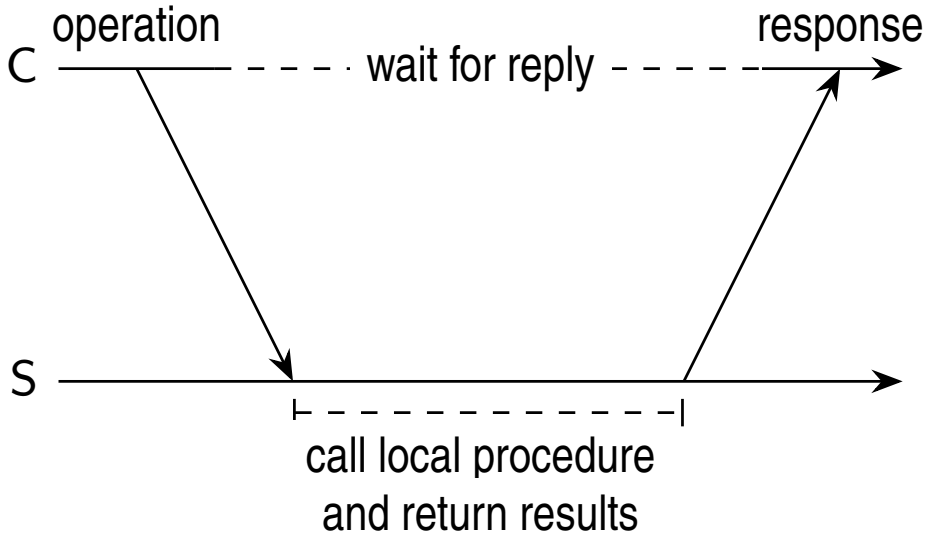
- Call-by-value (e.g., `data` if primitive, i.e., `int` or `bool`)
- Call-by-reference (e.g., `dbList`, most common in Python)
- Call-by-copy/restore:
  - variable copied to the stack by caller; then copied back after call, overwriting the caller's original value
  - not used in C, Python, and most programming languages



Basic RPC Operation

Client and server stubs

- RPC makes remote procedure calls *look as much as local ones*
- When **append** is performed, a **client stub** is called, which
  - packs parameters into a message
  - sends the message to server
- Calling and called procedures are *not aware of distribution*



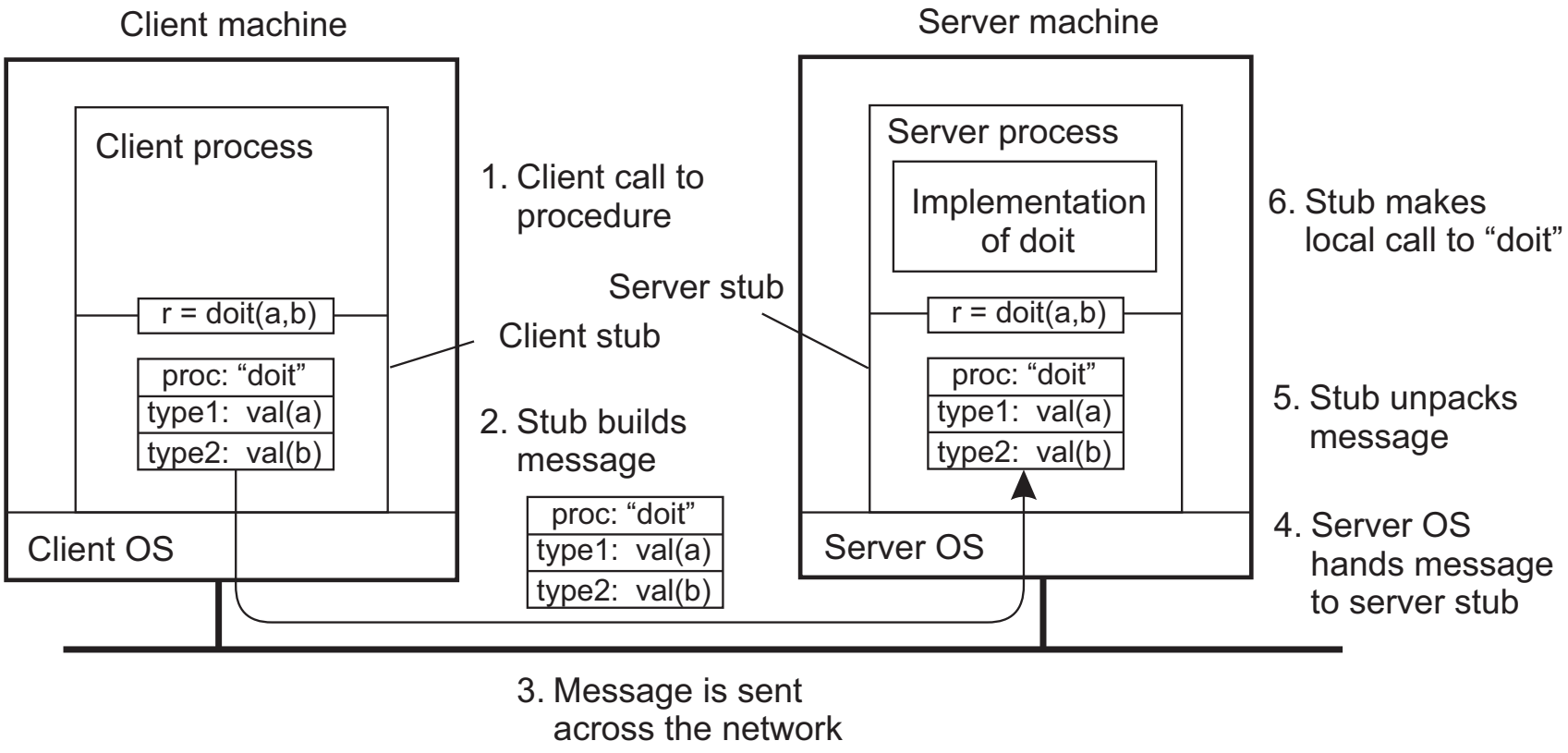
Principle of RPC between a client and server program.

RPC Steps

- client procedure calls the client stub in the normal way.
- client stub builds a message and calls the local operating system.
- client's OS sends the message to the remote OS.
- remote OS gives the message to the server stub.
- server stub unpacks the parameters and calls the server.
- server does the work and returns the result to the stub.
- server stub packs it in a message and calls its local OS.
- server's OS sends the message to the client's OS.
- client's OS gives the message to the client stub.
- stub unpacks the result and returns to the client.

RPC Steps

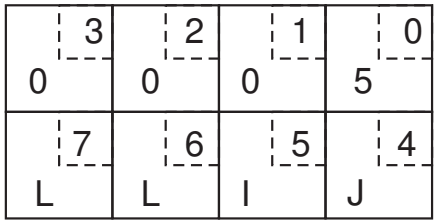
```
service DoIt { rpc doit (DoItReq) returns (DoItRep) {} }
message DoItReq { type1 a = 1; type2 b = 2; }
```



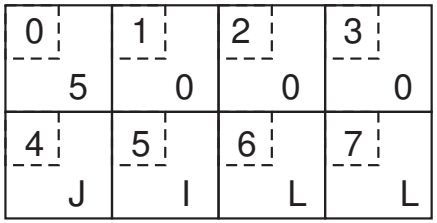
The steps involved in a remote computation through RPC

Parameter Passing

- Passing value parameters
  - What if client and server have different data representations?



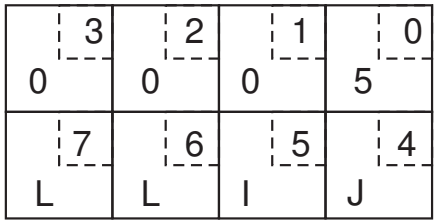
(a) The original message on Pentium



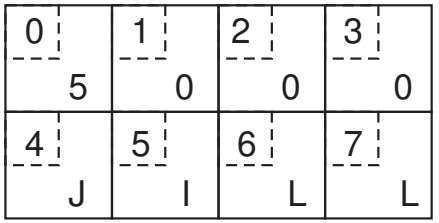
(b) The message after receipt on SPARK

Parameter Passing

- Passing value parameters
  - What if client and server have different data representations?



(a) The original message on Pentium

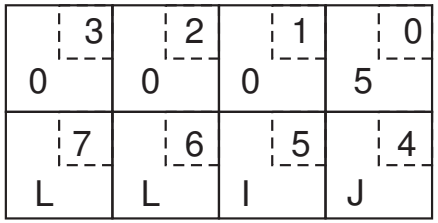


(b) The message after receipt on SPARK

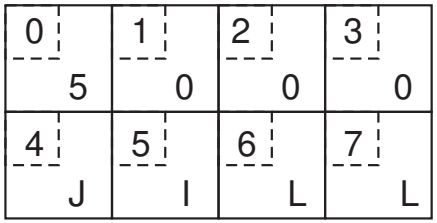
- Passing reference parameters
  - How are references (pointers) passed?
  - One solution is to copy the data structure to the server and back

Parameter Passing

- Passing value parameters
  - What if client and server have different data representations?



(a) The original message on Pentium

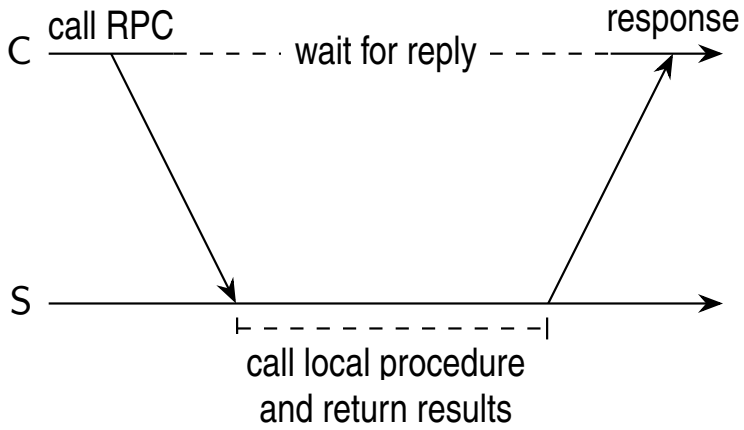


(b) The message after receipt on SPARK

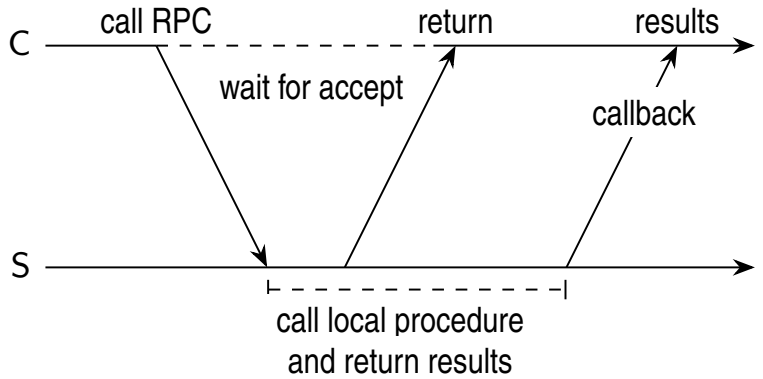
- Passing reference parameters
  - How are references (pointers) passed?
  - One solution is to copy the data structure to the server and back
- Parameter specification and stub generation
  - Interface Definition Language (IDL)
  - To simplify client-server applications based on RPC
  - Procedure interface specified in IDL and then compiled into a client stub and a server stub

Asynchronous RPC

If procedure call returns no value, client can continue.

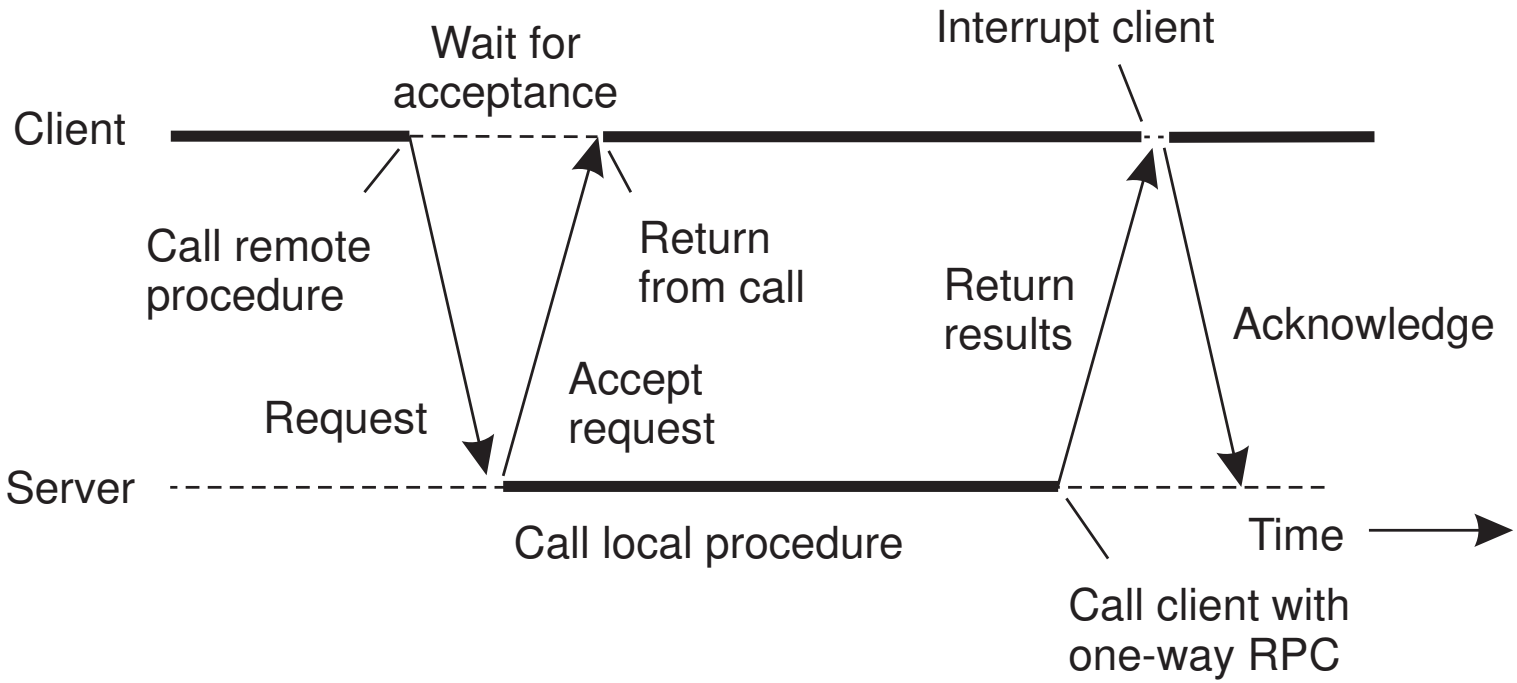


(a) The interaction between client and server in a traditional RPC



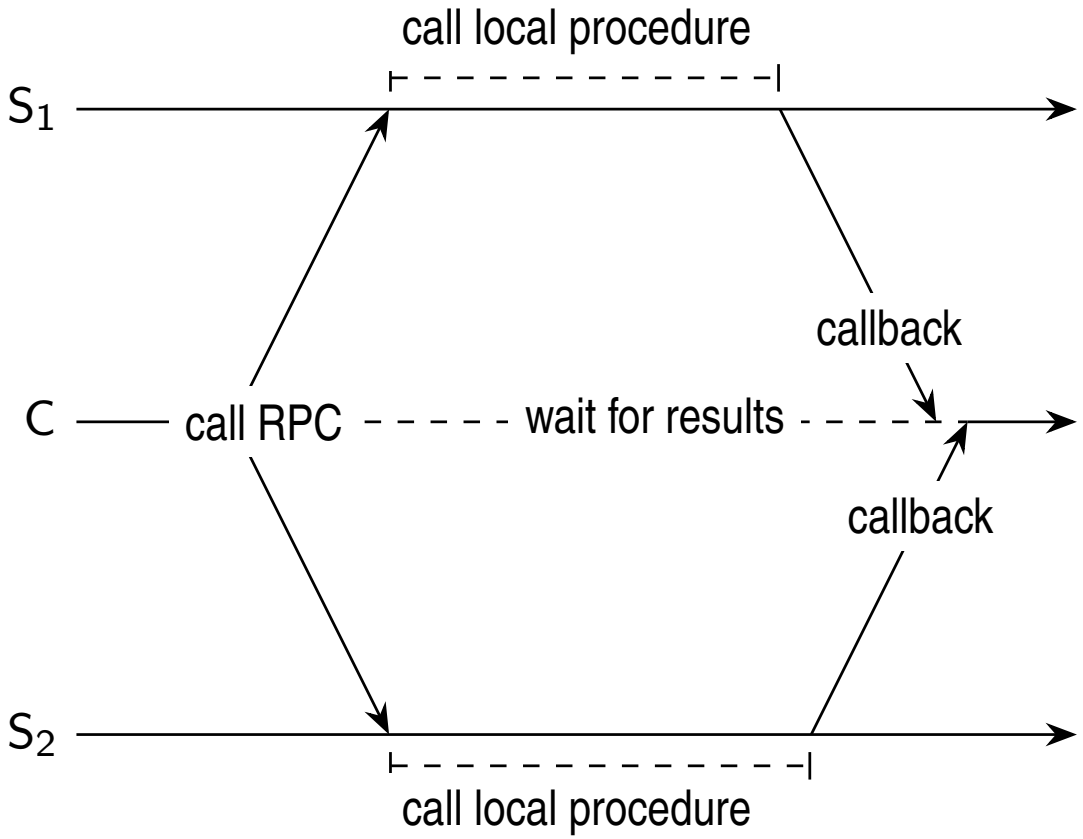
(b) The interaction using asynchronous RPC

Asynchronous RPC



A client and server interacting through two asynchronous RPCs

Multicast RPC



The principle of a multicast RPC

Message-Oriented Communication

Message-oriented transient communication

Berkeley sockets

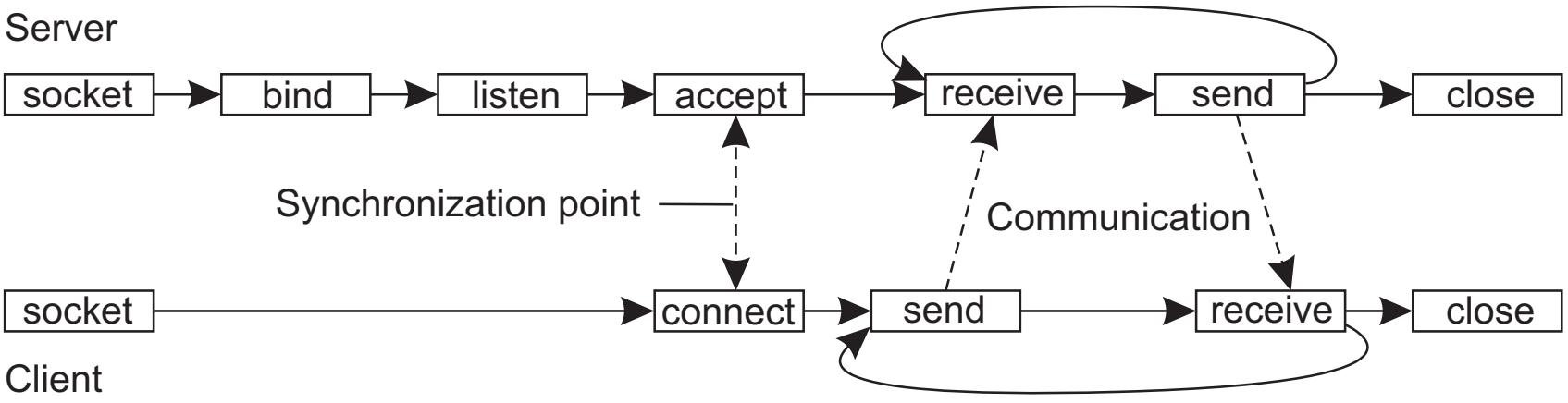
- **Socket**: communication end-point used by applications
- OS reserves resources for accommodating communication

Operation	Description
`socket`	Create a new communication end point
`bind`	Attach a local address to a socket
`listen`	Tell operating system what the maximum number of pending connection requests should be
`accept`	Block caller until a connection request arrives
`connect`	Actively attempt to establish a connection
`send`	Send some data over the connection
`receive`	Receive some data over the connection
`close`	Release the connection

The socket primitives for TCP/IP.

Message-oriented transient communication

Berkeley sockets



Connection-oriented communication pattern using sockets.

Message-oriented transient communication

Message-Passing Interface (MPI)

- More appropriate abstraction for parallel applications (vs sockets)
- Designed for *high-speed interconnection networks*
- Does not consider process crash and network partitions

Operation	Description
`MPI_BSEND`	Append outgoing message to a local send buffer
`MPI_SEND`	Send a message and wait until copied to local or remote buffer
`MPI_SSEND`	Send a message and wait until transmission starts
`MPI_SENDRECV`	Send a message and wait for reply
`MPI_ISEND`	Pass reference to outgoing message, and continue
`MPI_ISSEND`	Pass reference to outgoing message, and wait until receipt starts
`MPI_RECV`	Receive a message; block if there is none
`MPI_IRECV`	Check if there is an incoming message, but do not block

Some of the most intuitive message-passing primitives of MPI

Message-oriented persistent communication

Message-queueing models

- Applications communicate by inserting messages in queues
- Sender and receiver not necessarily active at the same time
- In principle, each application has its own private queue
- Messages transferred from one server to another until destination
- Guarantee of delivery at destination, at some unknown time

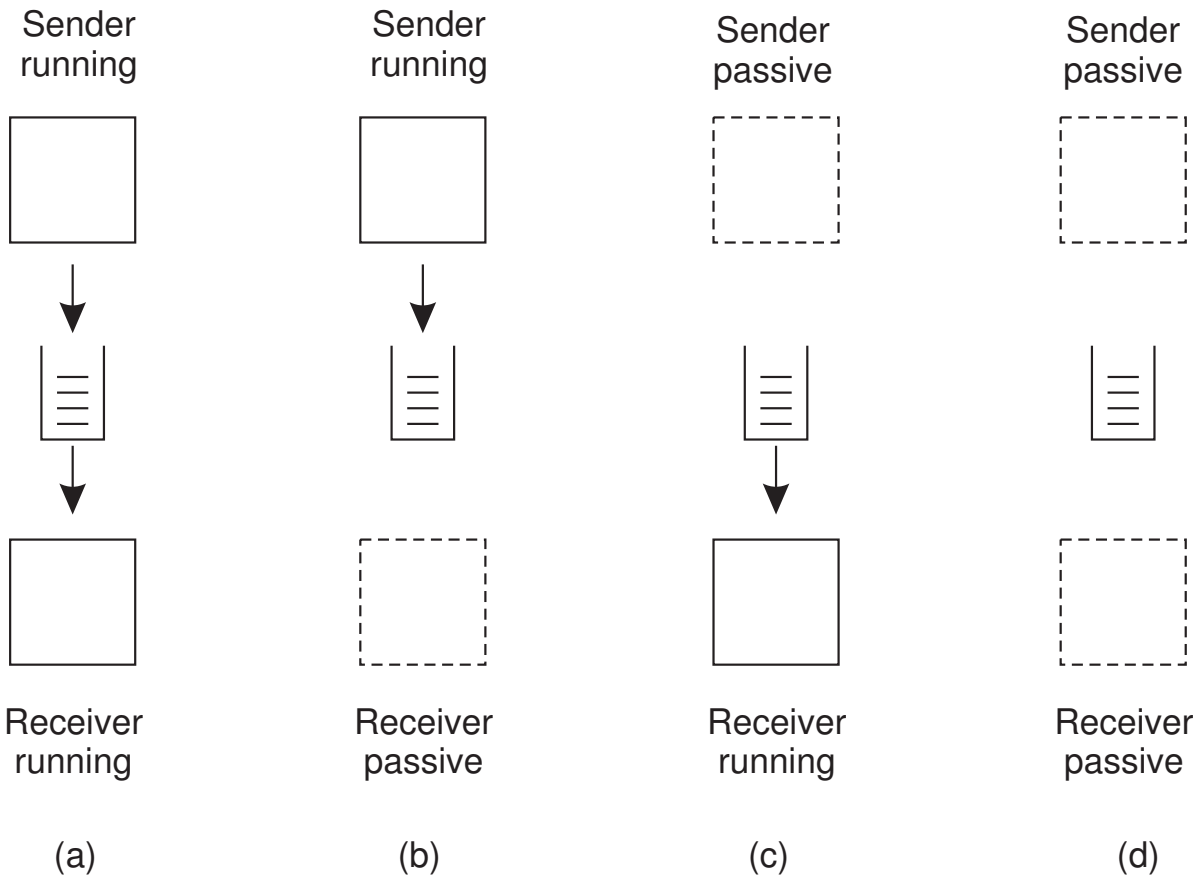
Simple interface:

Operation	Description
PUT	Append a message to a specified queue
GET	Block until the specified queue is nonempty, and remove the first message
POLL	Check a specified queue for messages, and remove the first. Never block
NOTIFY	Install a handler to be called when a message is put into the specified queue

Basic interface to a queue in a message-queuing system

Message-oriented persistent communication

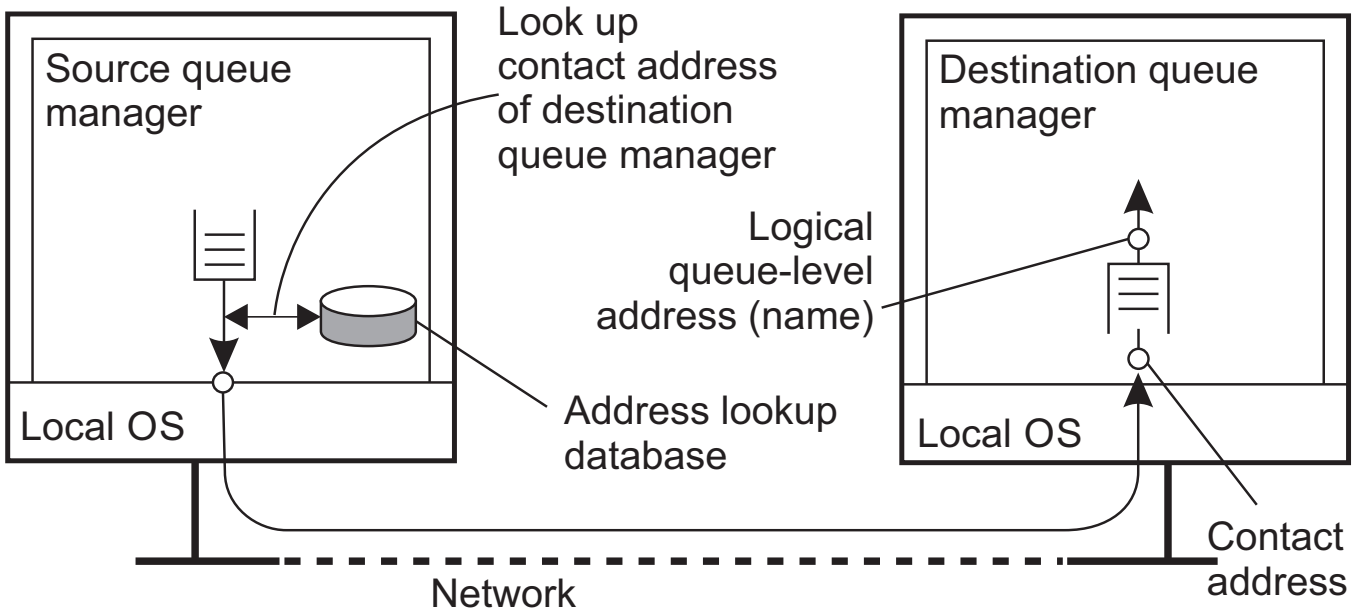
Message-queueing models



Four combinations for loosely-coupled communications using queues

General architecture of a MQ system

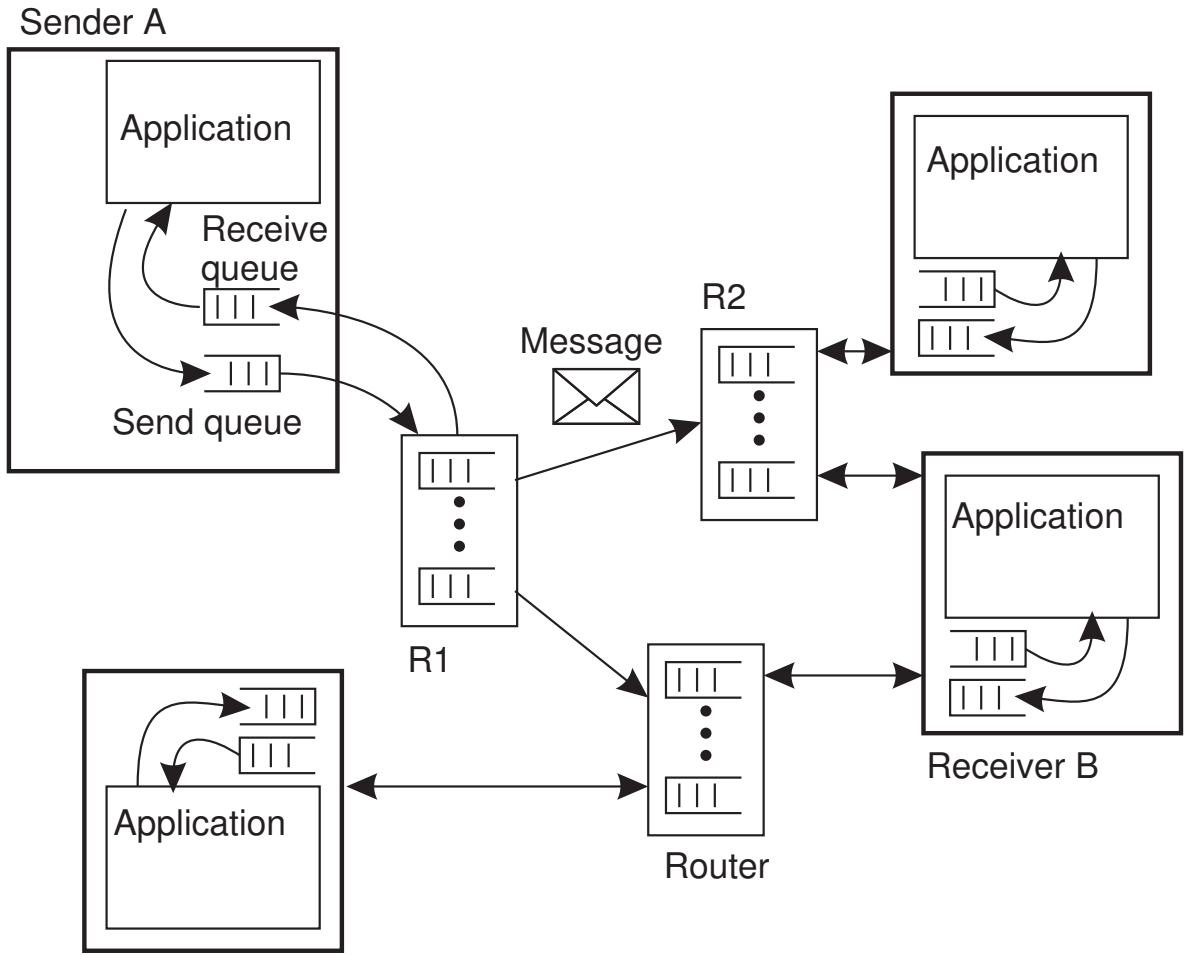
- Source queue: queue at the sender or nearby (same LAN)
- Messages contain destination queue
- Database of queue names to network locations



Relationship between queue-level addressing and network-level addressing

General architecture of a MQ system

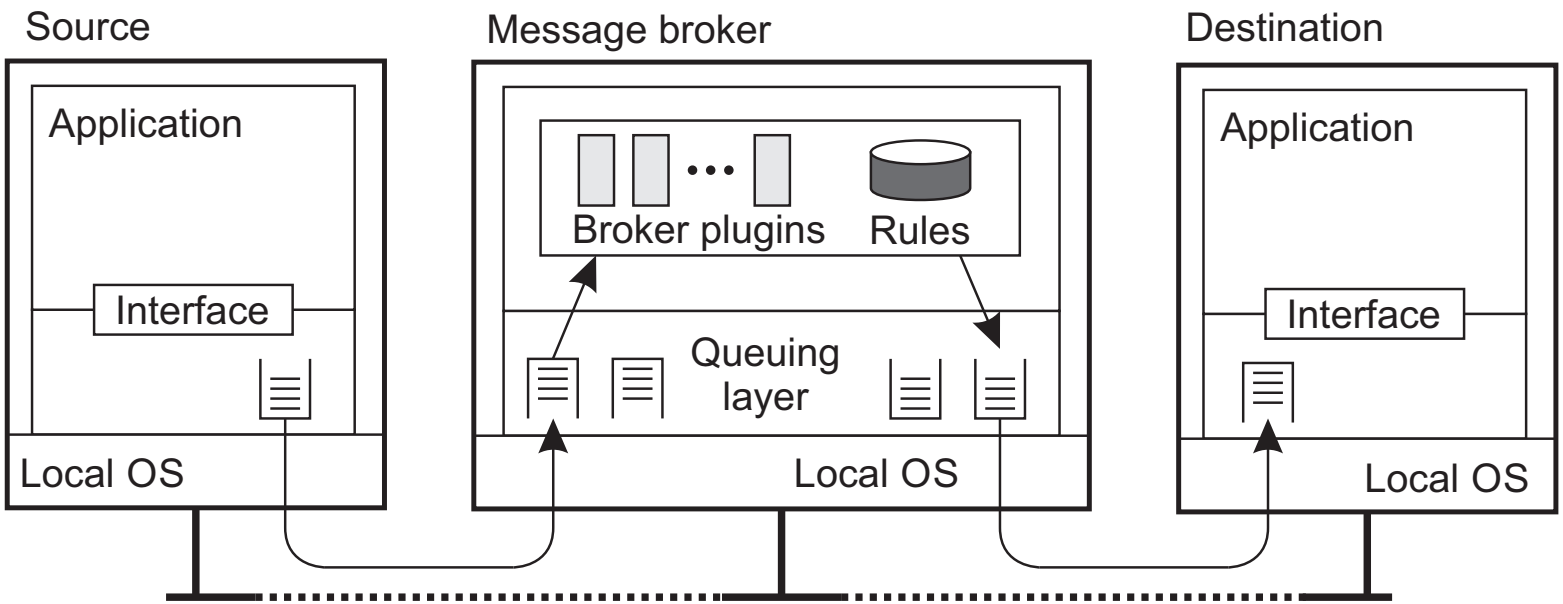
- Queue managers: interact with applications but also relays
- Relays forward messages to other queue managers
- Overlay network: composed of sender, destination, and relays



The general organization of a message-queuing system with routers

Message broker

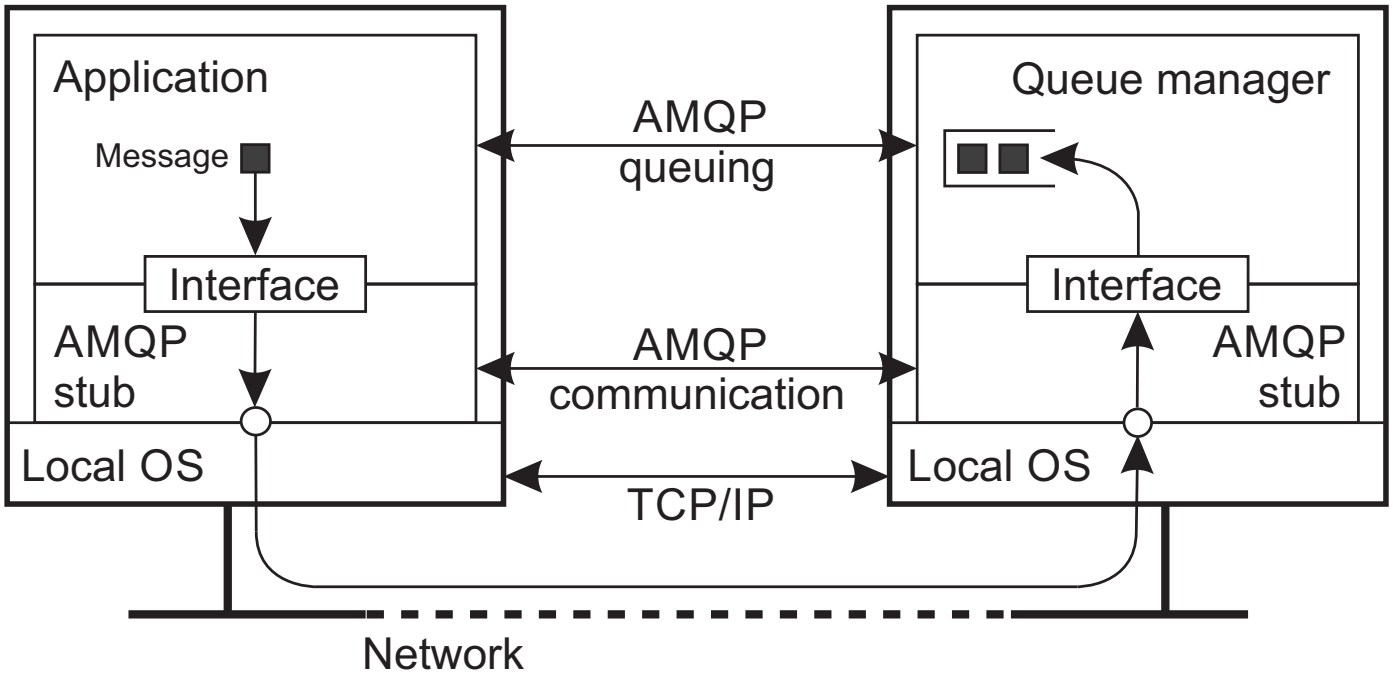
- Transform incoming messages to target format
- Very often act as an application gateway
- May provide subject-based routing capabilities



The general organization of a message broker

Advanced Message Queueing Protocol (AMQP)

Intended to play similar role as TCP



An overview of a single-server AMQP instance

- **Client** has a connection to queue manager
- **Queue manager** is a container of multiple one-way channels
- Two one-way channels can form a **session**
- **Link** is like a socket, maintains state about message transfers

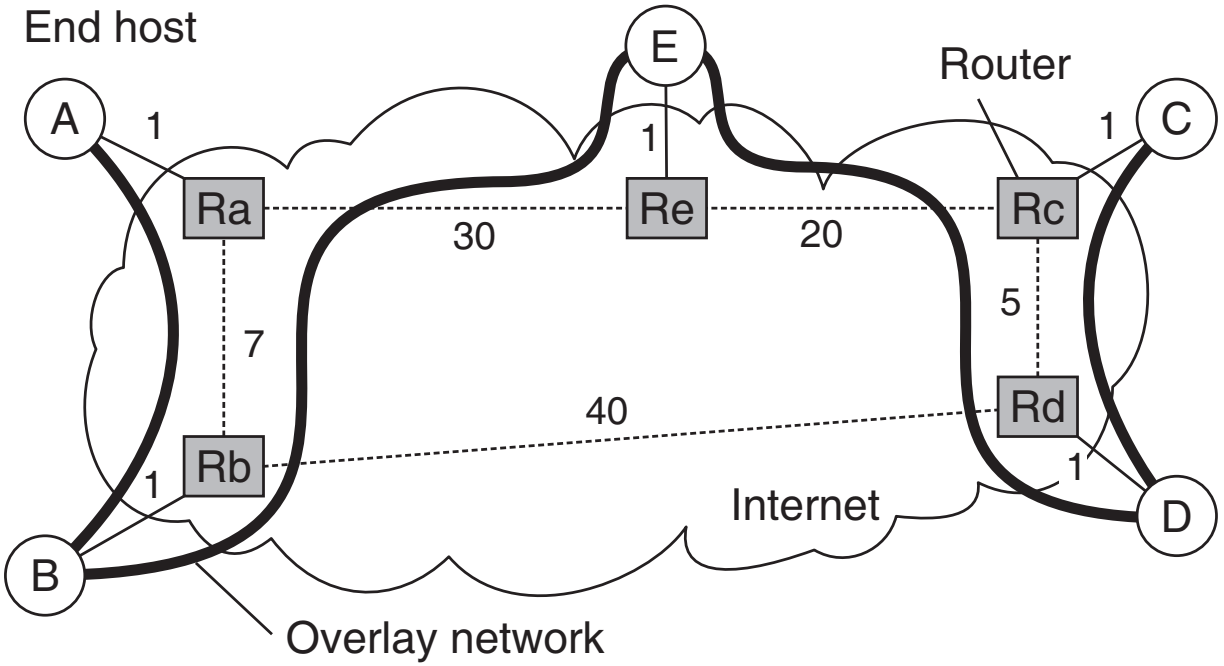
Multicast Communication

Application-level multicasting

- Support for sending data to multiple receivers
- Nodes organize into an **overlay network** at application level
- Two general approaches:
  - the overlay is a **tree**
    - single path between every pair of nodes
  - the overlay is a **mesh**
    - multiple paths between pairs of nodes
    - ✓ generally more robust

Application-level multicasting

Overlay construction



The relation between links in an overlay and actual network-level routes. Dark lines show inefficient way of broadcasting from A

- **Link stress:** how often a link is traversed by same message
- **Stretch:** ratio of delays between application- and network- levels

Application-level multicasting

Flooding

Process sends message `m` to *all neighbors*, each recipient does the same if not already seen `m`, ...