Concurrent processes and programming (cont'd)

Language mechanisms for synchronization

A concurrent language extended from a sequential language adds additional constructs to provide:

- Specification of concurrent activities
- Synchronization of processes
- Interprocess communication
- Nondeterministic execution of processes

Synchronization mechanisms and language facilities

Synchronization methods	*Language facilities*
Shared-variable synchronization	
semaphore	shared variable and system call
monitor	data type abstraction
conditional critical region	control structure
serializer	data type and control structure
path expression	data type and program structure
Message passing synchronization	
communicating sequential processes	input and output
remote procedure call	procedure call
rendezvous	procedure call and communication

Message passing synchronization

- The only means of communication in distributed systems
- Implicit synchronization: messages can be received only after they have been sent
- Non-blocking send, blocking receive: *asynchronous* message passing
- Blocking send, blocking receive: *synchronous* message passing

Asynchronous message passing:

- Is an extension of the semaphore concept to distributed systems
- Send operations assume that the channel has an unbounded buffer
- Example: **pipe** and **socket**

Synchronous message passing:

- No buffering of messages in the communication channel
- **rendezvous** between send and receive
- Examples: Communication Sequential Processes (CSP), Remote Procedure Call (RPC) - asymmetrical communication, Ada rendezvous - symmetrical communication

Interprocess communication and coordination

- Distributed IPC and process coordination are based on message passing
- Dependent on the ability to locate communication entities: role of the **name service**
- Three fundamental message passing communication models:
  - message passing
  - request/reply (RPC)
  - transaction communication
- Distributed process coordination examples:
  - distributed mutual exclusion
  - leader election

Message passing communication

- Messages are collections of data objects
- Their structure and interpretations are defined by the peer applications
- Communicating processes pass composed messages to the system transport service

interprocess communication	transaction request/reply (RPC) message passing
network operating system	transport connection
communication network	packet switching

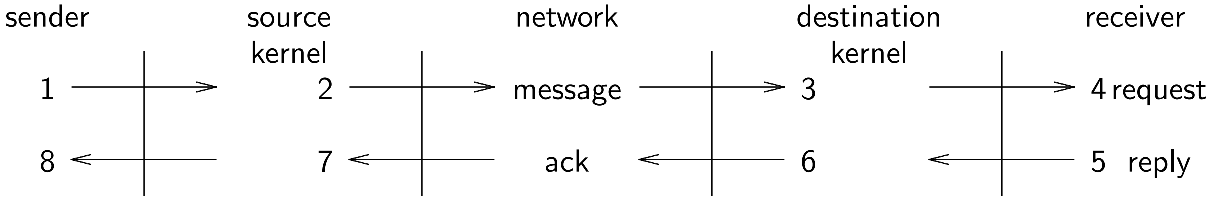
Basic communication primitives:

- `send(destination, message)`
- `receive(source, message)`

where `source` or `destination` = (process name, link, mailbox, port)

process name (global PID) - direct communication primitive
link (connection) - direct communication primitive
mailbox - indirect communication primitive many-to-many
port - indirect communication primitive many-to-one

Message synchronization and buffering:



1. **Nonblocking send**: 1+8
2. **Blocking send**: 1+2+7+8
3. **Reliable blocking send**: 1+2+3+6+7+8
4. **Explicit blocking send**: 1+2+3+4+5+6+7+8
5. **Request and reply**: 1-4, service, 5-8

At the receiving site **blocking** is quite explicit: blocked for message arrival

Implicit buffer space:

- in sender's kernel
- in receiver's kernel
- in the communication network

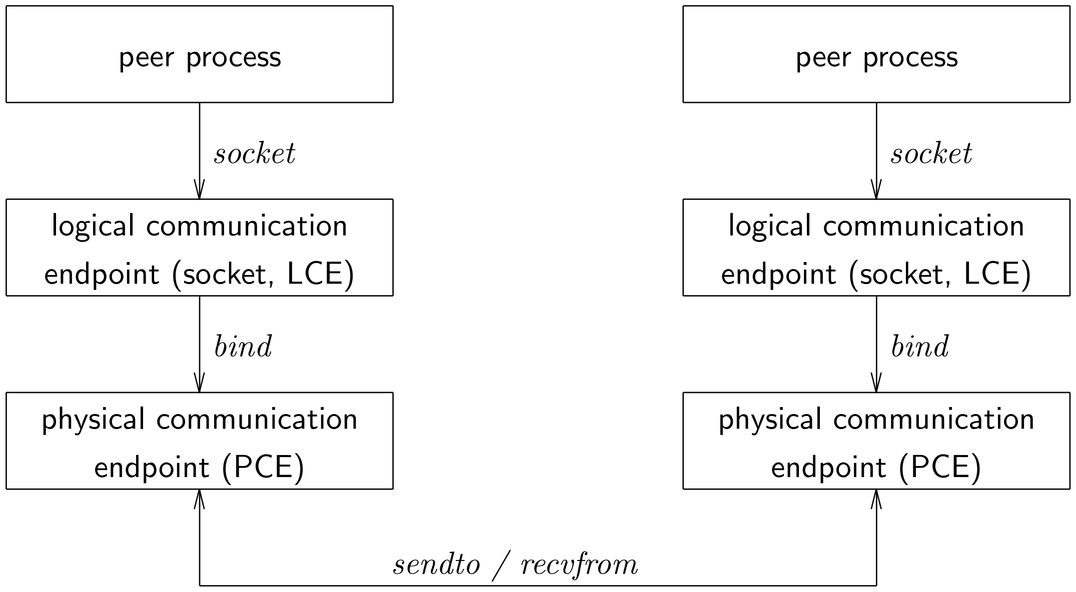
Pipe and socket APIs

- Used in both UNIX and Windows
- **Pipes**: implemented with finite-size, FIFO byte stream buffer maintained by the OS kernel
  - created with the `pipe` system call, which returns two descriptors (one for writing, one for reading)
  - data in pipes are uninterpreted byte sequences
  - are *anonymous*

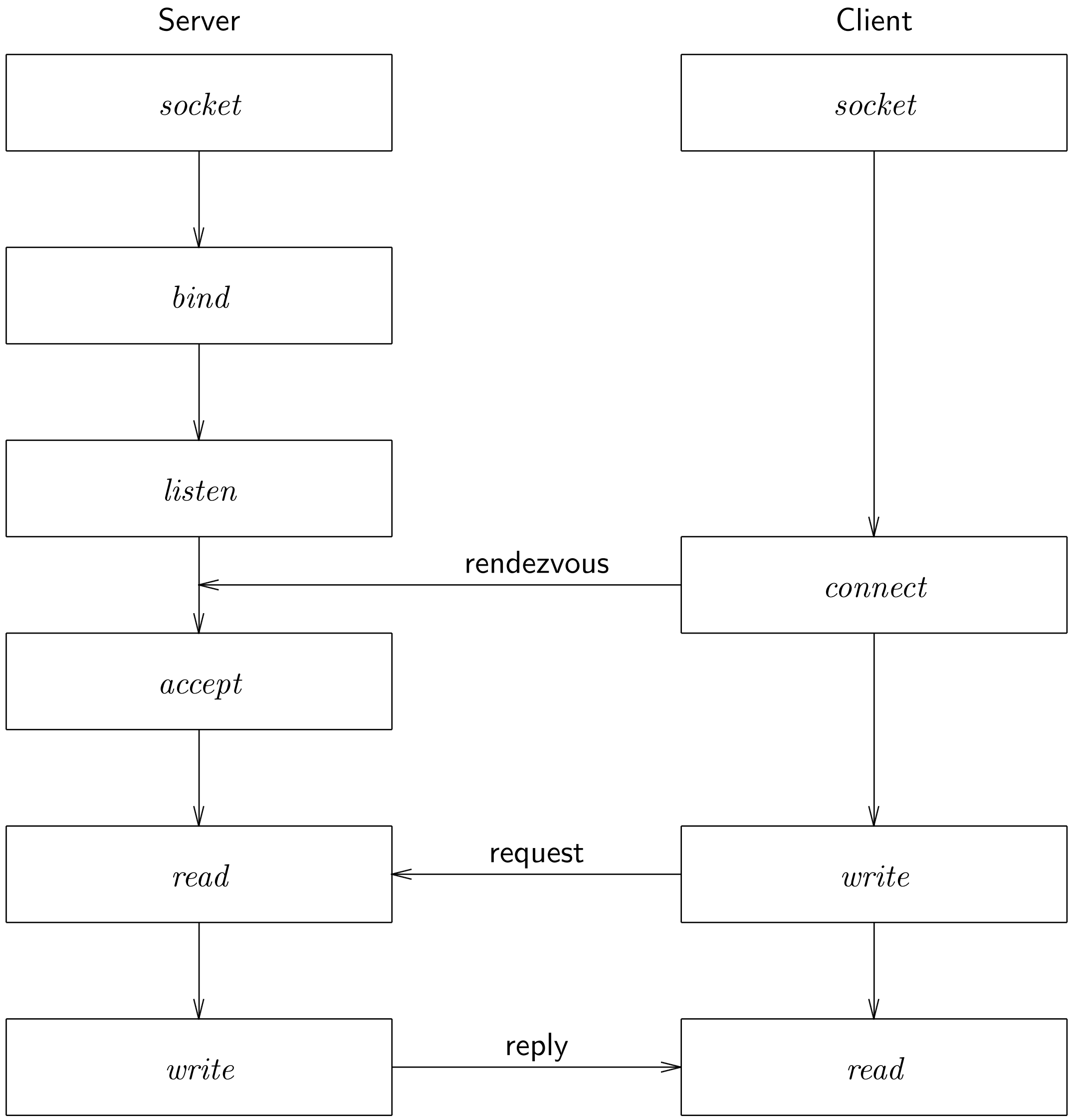
  - variation: *named* pipes - use the semantics of ordinary files for opening, communicating processes need not exist concurrently
  - use limited to a single domain within a common file system (except named pipes under Windows)

- **Socket** is a communication endpoint of a communication link managed by the OS's transport system
  - modeling network I/O based on conventional file I/O
  - created by the `socket` system call
  - used for file-oriented `read`/`write` operations
  - used for communication-specific `send`/`receive` operations
  - communicate over various network protocols, for example TCP, UDP, (raw) IP
  - socket descriptor is a logical communication endpoint (LCE); it must be associated with a physical communication endpoint (PCE): for example host network address and transport port in case of TCP or UDP

Connectionless socket communication:


peer process
socket
logical communication endpoint (socket, LCE)
bind
physical communication endpoint (PCE)
peer process
socket
logical communication endpoint (socket, LCE)
bind
physical communication endpoint (PCE)
sendto / recvfrom

Connection-oriented socket communication:



Secure Socket Layer

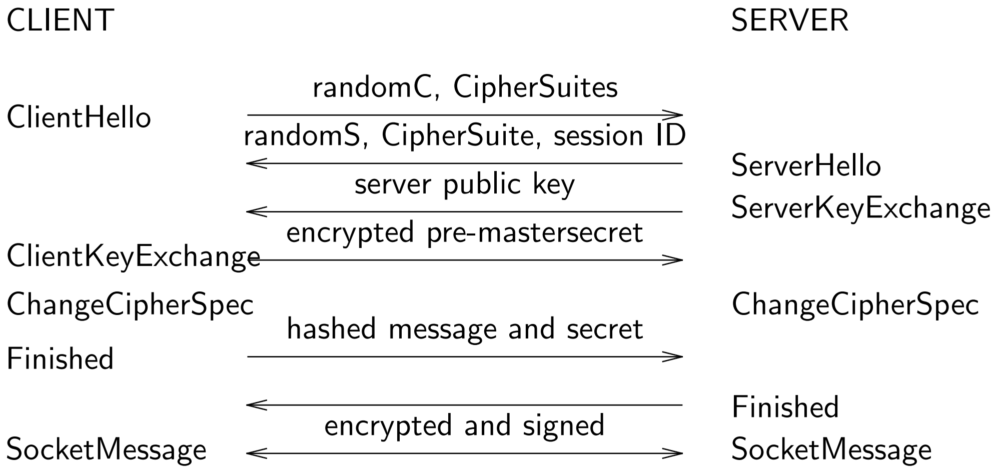
Goals:

- **Privacy** in socket communication
- **Integrity** of socket data
- **Authenticity** of servers and clients using asymmetric public-key cryptography

SSL consists of two protocols:

- Handshake protocol
  - establishing the **write keys** and **MAC secret** (message authentication check) → **master secret**
  - Validating the authenticity of clients and servers
  - Client of the Record Layer protocol
- Record Layer protocol
  - Fragmentation, compression/decompression
  - Encryption/decryption of message records

SSL Handshake protocol



CLIENT SERVER

ClientHello — randomC, CipherSuites →

randomS, CipherSuite, session ID ← ServerHello

server public key ← ServerKeyExchange

ClientKeyExchange — encrypted pre-mastersecret →

ChangeCipherSpec ChangeCipherSpec

Finished — hashed message and secret →

← Finished

SocketMessage ↔ encrypted and signed ↔ SocketMessage