Client-Server Model

- **Key Idea:** Structure a distributed system as a collection (or group) of cooperating processes, called servers, that offer services to users, called clients.

- Examples?
Design Issues

- Addressing
- Blocking
- Buffering
- Reliability
Addressing

How does a client know/find a server's address?

1. Hardwire into client code: `machine.local-id`
   
   Disadvantage: Not transparent

2. Randomly chosen sparse global process-id;
   locate via broadcast messages and cache for future use.
   Disadvantage: Potential cost

3. Lookup via a name server; logical id is associated with a server (or service).
   Disadvantage: centralized name server vs. consistency of replicas.

Blocking vs. Nonblocking

- **Blocking (or synchronous):**
  Sender blocks until the message is sent.
  Receiver is suspended until the message is received.

- **Nonblocking (or asynchronous):**
  Control is returned to caller immediately.

- Is the socket send call in UNIX blocking or nonblocking?

- Potential problems with nonblocking send?
  
  `Sender cannot modify message buffer!`

  - Nonblocking with copy to kernel
  - Nonblocking with interrupt
Blocking vs. Nonblocking

- Choices for receive?
  - Blocking
  - Conditional receive
  - Blocking with timeout
  - Nonblocking (tell kernel where to store msg.) with polling vs. interrupt

- Multi-threaded systems: have one thread block while others continue
Buffered vs. Unbuffered

What to do when a message arrives and the receiver has not made a call?

1. Unbuffered primitive; kernel discards message.
2. Kernel buffers (with timeout) all messages without knowing who may need it.
3. Mailbox approach: receiver tells the kernel to store the messages on a particular network address (port) in a special data structure.

What if a mailbox is full?

Discard message or use flow control to prevent sender from sending more messages until ack arrives.

Reliable vs. Unreliable

What about message loss?
No guarantee that the message is delivered after being sent by the client even in the synchronous case.

- Unreliable send semantics (application/user responsibility)
- Ack messages at kernel level
  1. Request: C → S
  2. Ack: K → K
  3. Reply: S → C
  4. Ack: K → K
- Exploit C/S model
  1. Request: C → S
  2. Reply: S → C
  3. Ack: K → K
Other Implementation issues

- Bounded packet size/split message into packets
- Ack every packet vs. ack every message
- Buffer full? try again.
- Server not responding? Are-you-alive message.
- Multi-threaded vs. single-threaded implementation.