Communication Models in Distributed Systems

- **Peer-to-peer**
  - Producer / consumer

- **Client / server**
  - Clients ask dedicated server to perform some specific service

- **Central coordinator** (many-to-one)
  - Nodes send information to coordinator; coordinator makes decision
  - Central point of failure

- **Distributed consensus** (one-to-many)
  - Nodes send information to each other; group as a whole reaches a consensus
  - Large amount of communication required

Client / Server Model using Message Passing

- **Client / server model**
  - **Server** = process (or collection of processes) that provides a service
    - Example: name service, file service
  - **Client** — process that uses the service
  - Request / reply protocol:
    - Client sends request message to server, asking it to perform some service
    - Server performs service, sends reply message containing results or error code

Message Passing using Send & Receive

- **Blocking send**:
  - send(destination-process, message)
  - Sends a message to another process, then blocks (i.e., gets suspended by OS) until message is received

- **Blocking receive**:
  - receive(source-process, message)
  - Blocks until a message is received (may be minutes, hours, ...)

- **Producer-Consumer problem**:
  ```
  /* producer */
  repeat forever
  ... produce item nextp
  ... send(consumer, nextp)
  end repeat

  /* consumer */
  repeat forever
  receive(producer, nextc)
  ... consume item nextc
  ... end repeat
  ```

Direct vs. Indirect Communication

- **Direct communication** — explicitly name the process you’re communicating with
  - send(destination-process, message)
  - receive(source-process, message)
  - Variation: receiver may be able to use a “wildcard” to receive from any source
  - Receiver cannot distinguish between multiple “types” of messages from sender

- **Indirect communication** — communicate using mailboxes (owned by receiver)
  - send(mailbox, message)
  - receive(mailbox, message)
  - Variation: ... “wildcard” to receive from any source into that mailbox
  - Receiver can distinguish between multiple “types” of messages from sender
  - Some systems use “tags” instead of mailboxes
Buffering

- Link may be able to temporarily queue some messages during communication

- Zero capacity: (queue of length 0)
  - Blocking communication
  - Sender must wait until receiver receives the message — this synchronization to exchange data is called a rendezvous

- Bounded capacity: (queue of length $n$)
  - If receiver's queue is not full, new message is put on queue, and sender can continue executing immediately
  - If queue is full, sender must block until space is available in the queue

- Unbounded capacity: (infinite queue)
  - Non-blocking communication
  - Sender can always continue

Non-blocking Send & Receive

- Non-blocking send:
  - Sends, then continues execution without waiting for message to be received
  - Advantage: sending process can execute in parallel with message transmission
  - Problem: must avoid modifying message buffer until message has been received
    1. Keep message in user space, have kernel send interrupt when message has been received (difficult to program)
    2. Have program test for message receipt (works in some situations)

- Non-blocking receive:
  - Receive returns with buffer, but doesn’t know if there’s a message there or not
  - Must poll or receive interrupt when message is ready and process should perform a receive (difficult to program)

Failure Handling in Client / Server Communication

- Potential failures:
  - Loss of request
    - Server never performs request
  - Loss of response message
    - Client doesn’t know server performed request
  - Server may die or become unreachable
    - Did server perform request or not?

- 3-message reliable protocol:
  - Client sends request; blocks
  - Server sends reply; blocks
  - Client unblocks, sends acknowledgment; server unblocks

- 2-message protocol:
  - Client sends request; blocks
  - Server sends reply; client unblocks

Semantics in Presence of Failure (Client Can’t Locate Server, Lost Request)

- Client can’t locate server
  - Reasons: server down, new version of server code
  - Can’t just return error code always
  - Raise an exception (if supported)

- Lost request
  - Start timer after issuing request
    - If time expires, send request again
  - No problem if request was really lost
Semantics in Presence of Failure (Lost Request (cont.))

- Lost / delayed reply
  - OK to retransmit request only if remote procedure is idempotent (calling it multiple times is same as calling it once)
- If not idempotent, be more conservative:

Semantics in Presence of Failure (Error Recovery — Sequence Numbers)

- More general solution: attach a sequence number to every request and reply

Semantics in Presence of Failure (Server Crash)

- Possible scenarios
  - Request arrives, server crashes
  - Request arrives, request processed, server crashes
  - Request arrives, request processed, reply sent, server crashes
  - Desired response is different for each, but neither client nor server knows what it is

- Three (unattractive) alternatives:
  - Client keeps trying until it gets a response
    - Action carried out at least once
  - Client gives up and reports failure
    - Action carried out at most once (but maybe not at all)
  - Whatever…
    - No guarantees at all… easy to implement!
  - Ideal (unachievable)
    - Action carried out exactly once