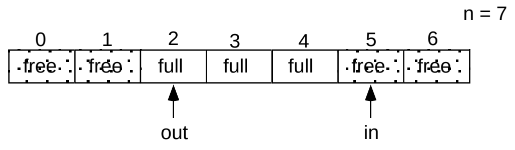
## Communication Models in Distributed Systems

- 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

## The Producer-Consumer Problem

- One process is a producer of information; another is a consumer of that information
- Processes communicate through a bounded (fixed-size) circular buffer

```
var buffer:  array[0..n-1] of items;     /* circular array */
in = 0
out = 0
```



```
/* producer */                        /* consumer */
repeat forever                        repeat forever
    …                                     while (in == out)
    produce item nextp                        do nothing
    …                                     nextc = buffer[out]
    while (in+1 mod n == out)             out = out+1 mod n
        do nothing                        …
    buffer[in] = nextp                    consume item nextc
    in = in+1 mod n                       …
end repeat                            end repeat
```

## 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 and acknowledged
- Blocking receive:
  - receive(*source-process*, *message*)
  - Blocks until a message is received (may be minutes, hours, …)
- Producer-Consumer problem:

```
/* producer */                  /* consumer */
repeat forever                  repeat forever
    …                               receive(producer,nextc)
    produce item nextp              …
    …                               consume item nextc
    send(consumer, nextp)           …
end repeat                      end repeat
```

## Non-blocking Send & Receive

- Non-blocking send:
  - Sends, then goes on to next instruction without waiting for an acknowledgment
  - Advantage: sending process can execute in parallel with message transmission
  - Problem: must avoid modifying message buffer until message has been received (but how do you know?)
    1. Copy message from user space to kernel space, then let process continue
    2. Keep message in user space, have kernel send interrupt when message has been received (difficult to program)
- 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)

## Buffering

- Link may have some capacity that determines the number of message that can be temporarily queued in it
- Zero capacity: (queue of length 0)
  - No place in link for messages to wait
  - Sender must wait until receiver is ready to receive the message
    - Sender blocks, waits for receiver to say it's ready, then resends message
    - Timeout mechanism is used to resend message, wait for acknowledgment
- Single-message capacity: (queue of length 1)
  - Simple method for synchronous communication
  - If receiver isn't ready, message is buffered

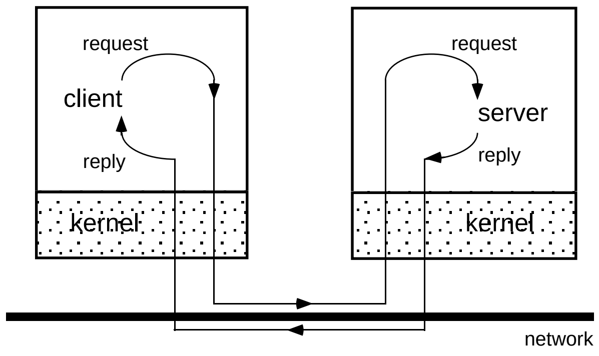
## Buffering (cont.)

- 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, either:
    - Send must return an error (leaves error handling up to programmer)
    - Sender must block until space is available in the queue (may result in deadlock)
- Unbounded capacity: (infinite queue)
  - Sender can always continue
  - Not possible in practice

## Direct vs. Indirect Communication

- Direct communication — explicitly name the process you're communicating with
  - send(*destination-process*, *message*)
  - receive(*source-process*, *message*)
  - Link is associated with exactly two processes
    - Between any two processes, there exists at most one link
    - The link may be unidirectional, but is usually bidirectional
- Indirect communication — communicate using mailboxes (owned by receiver)
  - send(*mailbox*, *message*)
  - receive(*mailbox*, *message*)
  - Link is associated with two or more processes that share a mailbox
    - Between any two processes, there may be a number of links
    - The link may be either unidirectional or bidirectional

## Client / Server Model using Message Passing



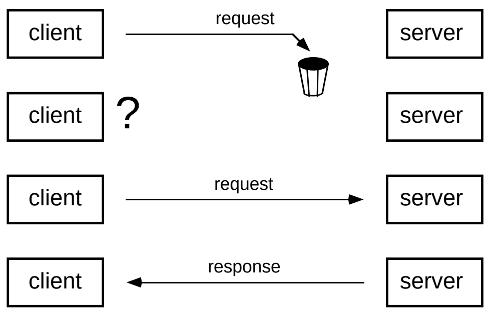
- Client / server model
  - *Server* = process (or collection of processes) that provides a *service*
    - Example: name service, file service
  - *Client* — process that uses the service
  - Request / reply protocol:
    - Client sends **request** message to server, asking it to perform some service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

## Failure Handling in Client / Sever 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

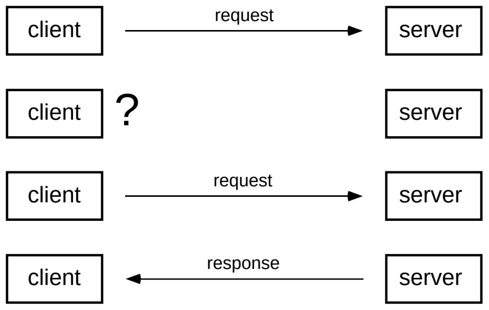
## Semantics in Presence of Failure (Client Can't Locate Server, Lost Request)

- 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

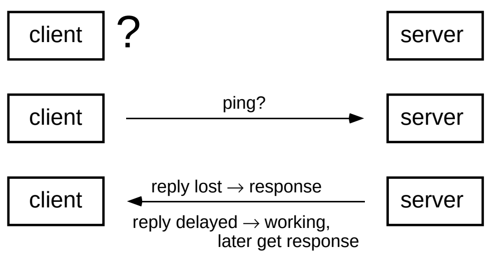


## Semantics in Presence of Failure (Lost Request (cont.))

- Lost / delayed reply
  - OK to retransmit request **only** if remote procedure is *idempotent* (calling it multiple times is same as calling it once)

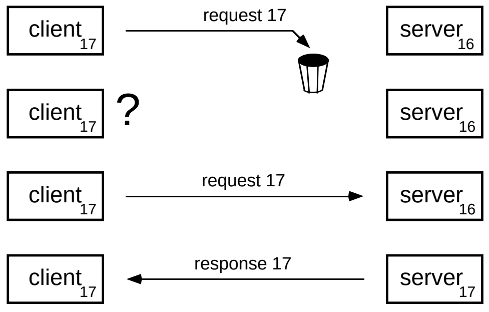


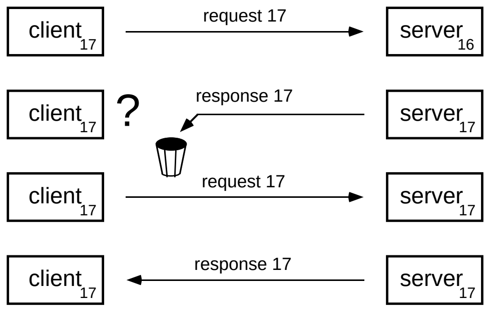
  - If not idempotent, be more conservative:



## 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*