Paradigms for Process Interaction in Distributed pgms.

Distributed computation — Concurrent pgms in which processes communicate by msg passing.

Process Interactions

1. networks of filters ✓ (merge sort)
2. clients & servers
3. heartbeat algms
4. probe/echo algms
5. broadcast algms
6. token-passing algms
7. decentralized servers
8. bags of tasks.

Message Passing

Decisions

- What kind of msgs
- When ?
- How

Different types of process

1. filters → "data transformer" (input channel) (output channel)
2. clients → triggering process
3. Servers → reactive process (non-terminating process)
4. peers → two peers

For Eg :-

Interprocess Communication

OS
1. Pipes
2. SHM
3. message passing
4. Semaphore
5. Monitors.

Programming Notation

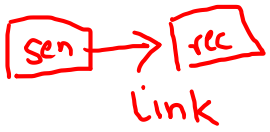
channel — physical comm^n
↳ Two primitives
(1) send
(2) receive

} global
↳ oneway / two-way

Design choices

(1) asynchronous message passing → non-blocking → different channels used for diff kind of msgs
(2) Synchronous message passing → blocking
(3) Generative communication → processes share a single comm^n channel tuple space
↳ naming association

(4) RPC (Remote Procedure call)
(5) Rendezvous



Chan input (char)

Chan ch ($f_1 : t_1, \ldots f_n : t_n$)

Channel name — name — type

channels are declared globally

```
chan input(char)          ← stream of characters
chan output([1:MAXLINE] char)   ← Line
Char_to_Line::
  var line[1:MAXLINE] : char
  var i : int := 1
  do true →
    receive input(line[i])
    do line[i] ≠ CR and i < MAXLINE →
    {line[1:i] contains last i input characters}
      i := i + 1; receive input(line[i])
    od
    send output(line); i := 1
  od
```

busy-wait polling

Send ch ($expr_1, \ldots expr_n$)

evaluate + append msg

unbounded

Send is nonblocking

receive → blocking primitive

Filters: A sorting network

- data transformer
- fun(input) → output
- "o/p of filter is a fn of its i/p"

SORT: $(\forall i: 1 \le i < n: sent[i] \le sent[i+1] \wedge$
values sent to *output* are a permutation of values received from *input*

An outline of *Sort* is as follows:

chan *input*(**int**), *output*(**int**)
Sort:: declarations of local variables
receive all numbers from *input*
sort the numbers
send the sorted numbers to *output*

{ 1 · · · · · n } [n+1]

MERGE:
in1 and *in2* are empty ∧ *sent*[*n* + 1] = EOS ∧
(∀*i*: 1 ≤ *i* < *n*: *sent*[*i*] ≤ *sent*[*i* + 1]) ∧
values sent to *out* are a permutation of values received from *in1* and *in2*

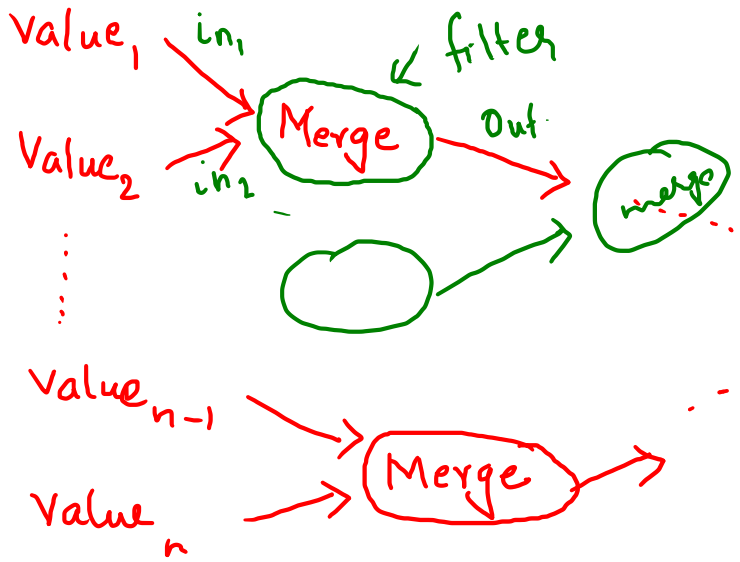
end sentinel

filter

```
                                  1          2          3
chan in1(int), in2(int), out(int)
Merge:: var v1, v2 : int
receive in1(v1); receive in2(v2)
do more input to process →
    send smaller of v1 and v2 to out
    receive another input value from
        in1 or in2
od
send out(EOS) {MERGE}
```

Sorting network

network of filters




```
chan in1(int), in2(int), out(int)
Merge:: var v1, v2 : int
        receive in1(v1); receive in2(v2)
        do v1 ≠ EOS and v2 ≠ EOS →
            if v1 ≤ v2 →
                send out(v1); receive in1(v1)
            [] v2 ≤ v1 →
                send out(v2); receive in2(v2)
            fi
        [] v1 ≠ EOS and v2 = EOS →
            send out(v1); receive in1(v1)
        [] v1 = EOS and v2 ≠ EOS →
            send out(v2); receive in2(v2)
        od
        send out(EOS) {MERGE}
```

exhaustive

Communication pattern → tree

Clients & Servers.

Centralized Server : Active Monitors → Condition variable
wait
signal
Procedures ←
↓
{mutual exclusion}

↓
Active
non-terminating

↓
passive
↓
shared m/y machines

```
monitor Mname   # Invariant MI
  var permanent variables
  initialization code
  proc op_1(formals_1) body_1 end
  .
  .
  .
  proc op_n(formals_n) body_n end
end
```

```
monitor Resource_Allocator
  { ALLOC:  avail ≥ 0  ∧  (avail > 0) ⇒ empty(free) }
  var avail : int := MAXUNITS, units : set of int,
      free : condition
  code to initialize units to appropriate values
  proc acquire( res id : int )
    if avail = 0 → wait(free)
    [] avail > 0 → avail := avail–1
    fi
    id := remove(units)
    end
  proc release( id : int )
    insert(id, units)
    if empty(free) → avail := avail+1
    [] not empty(free) → signal(free)
    fi
  end
end
```

Figure 2. Resource allocation monitor.

Static naming
dynamic naming

```
type op_kind = enum(ACQUIRE, RELEASE)
chan request(index : int, op_kind, unitid : int)
chan reply[1:n](int)
Allocator:: var avail : int := MAXUNITS, units : set of int, pending : queue of int
            var index : int, kind : op_kind, unitid : int
            code to initialize units to appropriate values
            {ALLOC: avail ≥ 0 ∧ (avail > 0) ⇒ empty(pending)}
            do true → receive request(index, kind, unitid)
                      if kind = ACQUIRE →
                          if avail > 0 →      # honor request now
                              avail := avail–1; unitid := remove(units)
                              send reply[index](unitid)
                          [] avail = 0 →      # save index of requesting process
                              insert(pending, index)
                          fi
                      [] kind = RELEASE →
                          if empty(pending) →     # release unitid
                              avail := avail+1; insert(units, unitid)
                          [] not empty(pending) →     # give unitid to first requester
                              index := remove(pending); send reply[index](unitid)
                          fi
                      fi
            od
Client[i: 1..n]:: var unitid : int
                  ...
                  # acquire a unit of the resource
                  send request(i, ACQUIRE, 0)      # unitid not needed
                  receive reply[i](unitid)
                  # use resource unitid and then later release it
                  send request(i, RELEASE, unitid)
```

Case 1

Case 2

Figure 3. Resource allocator and clients.

Monitor-Based Programs	*Message-Based Programs*
permanent variables	local server variables
procedure identifiers	*request* channel and operation kinds
procedure call	**send** *request*; **receive** *reply*
monitor entry	**receive** *request*
procedure return	**send** *reply*
wait statement	save pending request
signal statement	retrieve and process pending request
procedure bodies	arms of case statement on operation kind

Figure 4. Duality between monitors and centralized servers.

Disk scheduling

Shortest Seek time (SST)

- moving-head disk
- PA → cylinder no:-, track number, an offset
- whose cylinder is closest to the current position

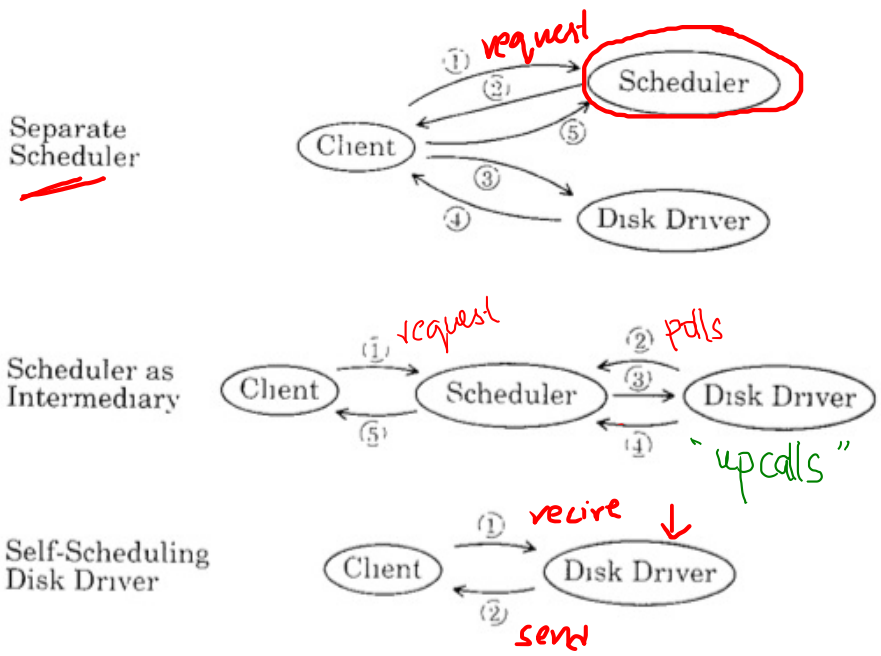
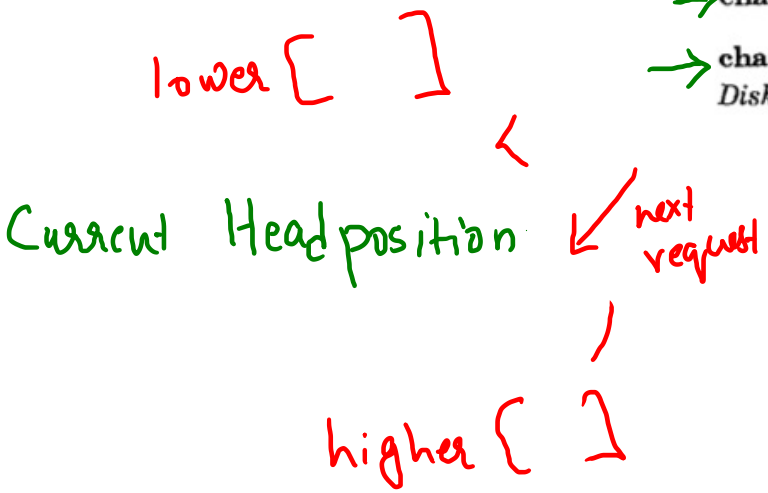


Figure 5. Disk scheduling structures with message passing.




```
chan request(index : int, cylinder : int, other argument types)
        # other arguments indicate read or write, disk block, memory buffer, etc.
chan reply[1:n](result types)
Disk_Driver:: var lower, higher : ordered queue of (int, int, other argument types)
        # contain index, cylinder, and other arguments of pending request
    var headpos : int := 1, nsaved := 0
    var index : int, cyl : int, args : other argument types
    { SST: lower is an ordered queue from largest to smallest cyl ∧
           all values of cyl in lower are ≤ headpos ∧
           higher is an ordered queue from smallest to largest cyl ∧
           all values of cyl in higher are ≥ headpos ∧
           (nsaved = 0) ⇒ (both lower and higher are empty) }
    do true →
        do not empty(request) or nsaved = 0 →
            # wait for first request or receive another one, then save it
            receive request(index, cyl, args)
            if cyl ≤ headpos := insert(lower, (index, cyl, args))
            [] cyl ≥ headpos := insert(higher, (index, cyl, args))
            fi
            nsaved := nsaved+1
        od
        # select best saved request; there is at least one in lower or higher
        if size(lower) = 0 → (index, cyl, args) := remove(higher)
        [] size(higher) = 0 → (index, cyl, args) := remove(lower)
        [] size(higher) > 0 and size(lower) > 0 →
            remove element from front of lower or higher depending
            on which saved value of cyl is closer to headpos
        fi
        headpos := cyl; nsaved := nsaved–1
        access the disk
        send reply[index](results)
    od
```

Figure 6. Self-scheduling disk driver.

Heart beat Algm

Pblm:- Each processor to determine the topology of entire n/w

Model

parallel

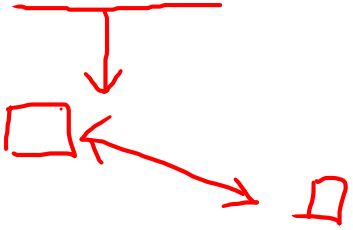
Processor → process

```
var top[1:n, 1:n] : bool := ([n*n] false)    # topology to be stored as set of all links
Node[p: 1..n]:: var links[1:n] : bool
                    # initialized so links[q] is true if q is a neighbor of Node[p]
                top[p, 1:n] := links     # fill in p'th row of top
                {top[p, 1:n] = links[1:n]}
```

Figure 8. Network topology using shared variables.

Communication links → Shared Channels

{graph DS}



Solution

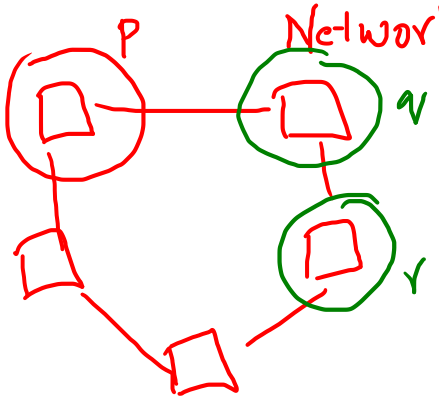
(1) Shared m/y acess

(2) distributed computation by replicate

heart beating : first expand, send info^n out

then contract, gather new info^n in

Network Topology :- Distributed Solution




```
chan topology[1:n](top[1:n, 1:n] : bool)      # one channel per node
Node[p: 1..n]:: var links[1:n] : bool [0,0,1,1,0]
        # initialized so links[q] is true if q is a neighbor of Node[p]
    var top[1:n, 1:n] : bool := ([n*n] false)      # local view of topology
    var r : int := 0
    var newtop[1:n, 1:n] : bool
    top[p, 1..n] := links      # fill in row for my neighbors
    { top[p, 1:n] = links[1:n] ∧ r = 0 }
    { ROUND: (∀ q: 1 ≤ q ≤ n: (dist(p, q) ≤ r) ⇒ top[q, *] filled in }
    do r < D →
        # send local knowledge of topology to all neighbors
        fa q := 1 to n st links[q] → send topology[q](top) af
        # receive their local topologies and or it with top
        fa q := 1 to n st links[q] →
            receive topology[p](newtop)
            { top := top or newtop }
        af
        r := r+1
    od
    { ROUND ∧ r = D } { TOPOLOGY }
```

Figure 9. Heartbeat algorithm for network topology; first refinement

Single process T

other nodes send its value of links

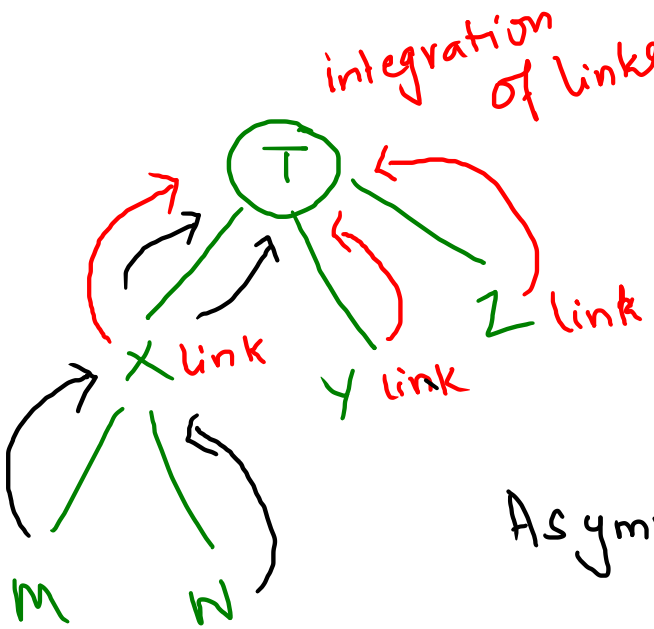
Drawback

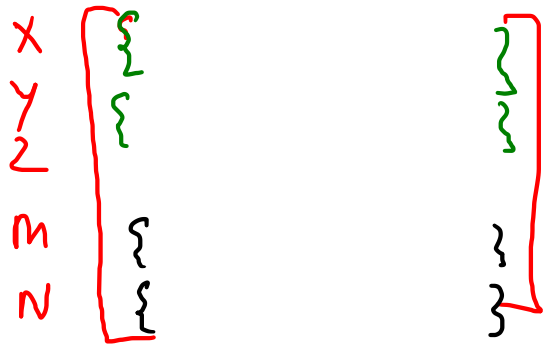
T is not one of the neighbor.

{Symmetric algm}

"Diameter of the network" → D

"distance b/w the farthest pair of nodes"





Asymmetric

Symmetric :- Each node needs to be able to compute for itself the entire topology

Node p → knows about the links to its neighbors

after one round of msgs

↳ p will have learned about topology within (two links) of it

after two rounds of msg

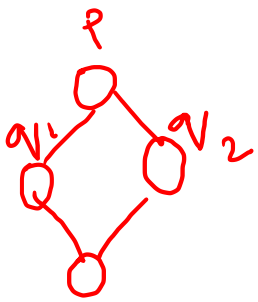
↳ each node will know about the topology within (3 links) of it.

⋮

r rounds

$$\{\forall q : 1 \le q \le n : dist(p,q) \le r$$

$$top[q, *] \text{ filled in}\}$$



dynamically

```
chan topology[1:n](sender : int, done : bool, top[1:n, 1:n] : bool)
Node[p: 1..n]:: var links[1:n] : bool
        # initialized so that links[q] true if q is a neighbor of Node[p]
    var active[1:n] : bool := links      # neighbors who are still active
    var top[1:n, 1:n] : bool := ([n*n] false)      # local view of topology
    var r : int := 0, done : bool := false
    var sender : int, qdone : bool, newtop[1:n, 1:n] : bool
    top[p, 1..n] := links      # fill in row for my neighbors
    {top[p, 1:n] = links[1:n] ∧ r = 0 ∧ ¬done}
    {ROUND ∧ (done ⇒ all rows in top are filled in)}
    do not done →
        # send local knowledge of topology to all neighbors
        fa q := 1 to n st links[q] → send topology[q](p, false, top) af
        # receive their local topologies and or it with top
        fa q := 1 to n st links[q] →
            receive topology[p](sender, qdone, newtop)
            top := top or newtop
            if qdone → active[sender] := false fi
        af
        if all rows of top have some true entry → done := true fi
        r := r+1
    od
    {ROUND ∧ all rows in top are filled in} {TOPOLOGY}
    # send topology to all neighbors who are still active
    fa q := 1 to n st active[q] → send topology[q](p, true, top) af
    # receive one message from each to clear up message queue
    fa q := 1 to n st active[q] → receive topology[p](sender, qdone, newtop) af
```

Figure 10. Heartbeat algorithm for network topology; final version.

Probe/Echo Algm.

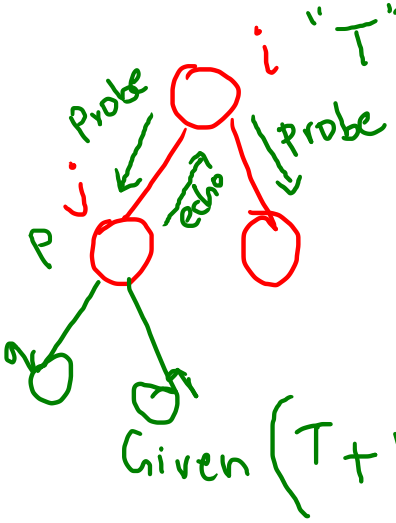
```
chan probe[1:n](span_tree[1:n, 1:n] : bool, message_type)
Node[p: 1..n]:: var span_tree[1:n, 1:n] : bool, msg : message_type
                receive probe[p](span_tree, msg)
                fa q := 1 to n st q is a child of p in span_tree →
                    send probe[q](span_tree, msg)
                af
Initiator:: var i : int := index of node that is to initiate broadcast
            var top[1:n, 1:n] : bool      # initialized with topology of the network
            var span[1:n, 1:n] : bool, msg : message_type
            compute spanning tree of top rooted at i and store it in span
            msg := message to be broadcast
            send probe[i](span, msg)
```

Figure 11. Broadcast using a spanning tree.

Tree, Graph

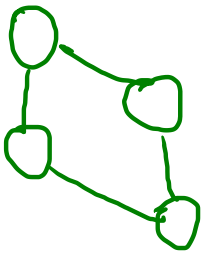
{DFS →}

Probe → a msg sent



broadcast

Given (T + msg)

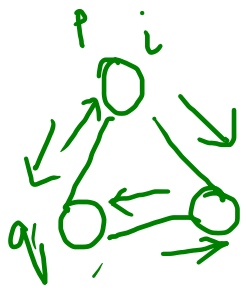


$G = (V, E)$

$G_{MST} = (V, E' \subset E)$

First node i
Sends the msg
to all its neighbor

"Form tree"




```
chan probe[1:n](message_type)
Node[p: 1..n]:: var links[1:n] : bool := neighbors of node p
                var num : int := number of neighbors of p, msg : message_type
                receive probe[p](msg)
                # send msg to all neighbors
                fa q := 1 to n st links[q] → send probe[q](msg) af
                # receive num−1 redundant copies of msg
                fa q := 1 to num−1 → receive probe[p](msg) af
Initiator:: var i : int := index of node that is to initiate broadcast
            var msg : message_type := message to be broadcast
            send probe[i](msg)
```

Figure 12. Broadcast using neighbor sets

```
const source = i      # index of node that initiates the algorithm
chan probe[1:n](sender : int)
chan echo[1:n](links[1:n, 1:n] : bool)      # contents are part of the topology
chan finalecho(links[1:n, 1:n] : bool)      # final echo to Initiator
Node[p: 1..n]:: var links[1:n] : bool := neighbors of node p
                var localtop[1:n, 1:n] : bool := ([n*n] false)
                localtop[p, 1:n] := links      # put neighbor set in localtop
                var newtop[1:n, 1:n] : bool
                var parent : int      # will be node from whom probe is received
                receive probe[p](parent)
                # probe on to to all other neighbors, who are p's children
                fa q := 1 to n st links[q] and q ≠ parent → send probe[q](p) af
                # receive echoes for all children and union them into localtop
                fa q := 1 to n st links[q] and q ≠ parent →
                      receive echo[p](newtop); localtop := localtop or newtop
                af
                if p = source → send finalecho(localtop)      # this node is the root
                [] p ≠ source → send echo[parent](localtop)
                fi
Initiator:: var top[1:n, 1:n] : bool      # network topology as set of links
            send probe[source](source)
            receive finalecho(top)
```

Figure 13. Probe/echo algorithm for topology of a tree.

Broadcast Algms

Processors = 'n'

{ send
  receive

Ch [1 : n] = array of channels

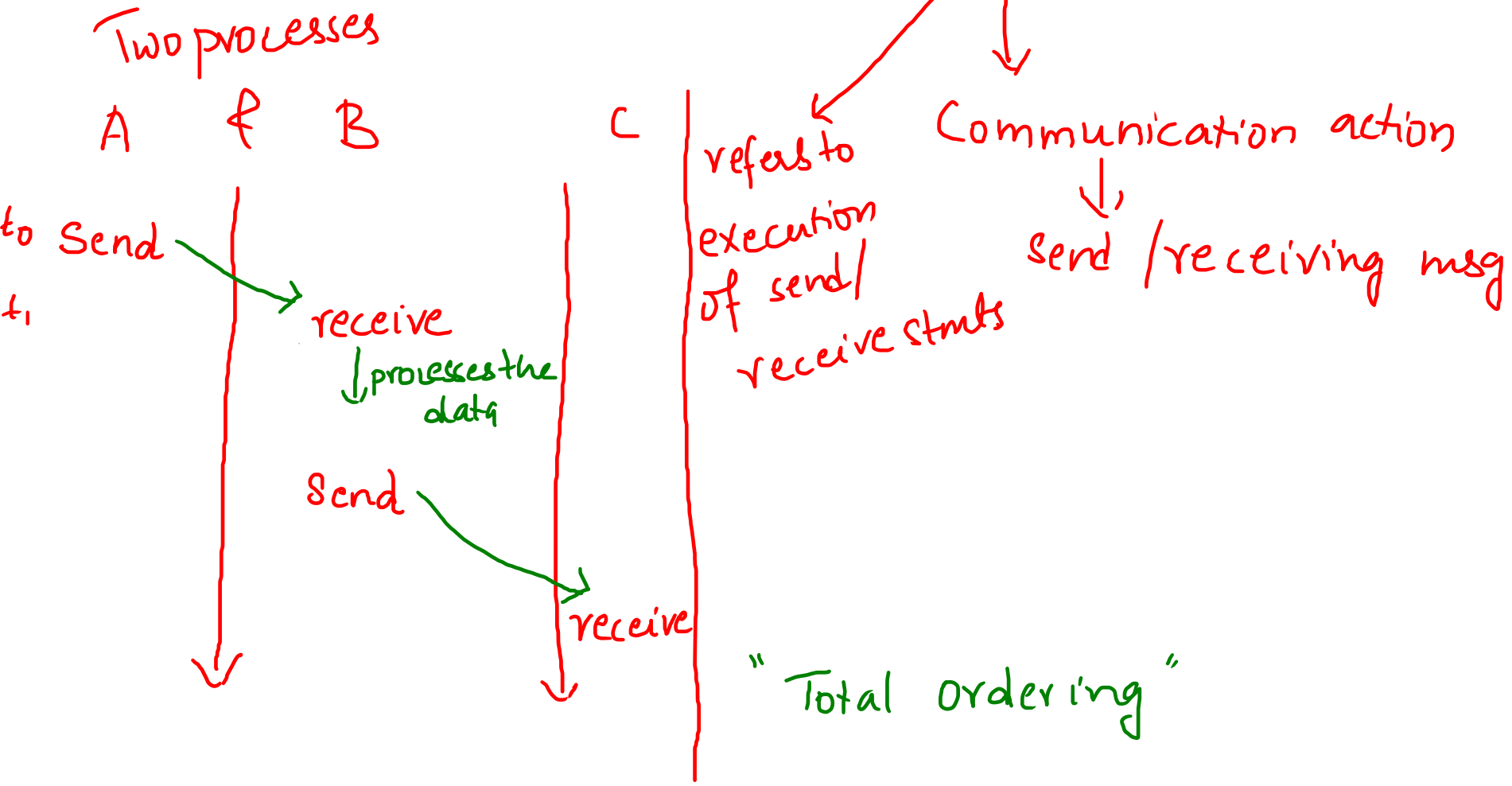
i^{th} process to
P[i]

{ broadcast ch(m) } → 'n' send messages

broadcast
↓
receive

Two processes
A & B
\ /
different orders

Logical clocks & Event Ordering



Logical clock

is an integer counter that is incremented when events occur.

Process = logical clock + msg contains a timestamp.

lc in process A.

(1) Send — {current value of lc} (timestamp) and increment lc by 1

(2) receive — lc = max of (lc, ts + 1) then increment lc by 1

Distributed Semaphores

P(s) ↓ { if s is true decrements (s) }

V(s) ↓ Increment s

Semaphore invariant

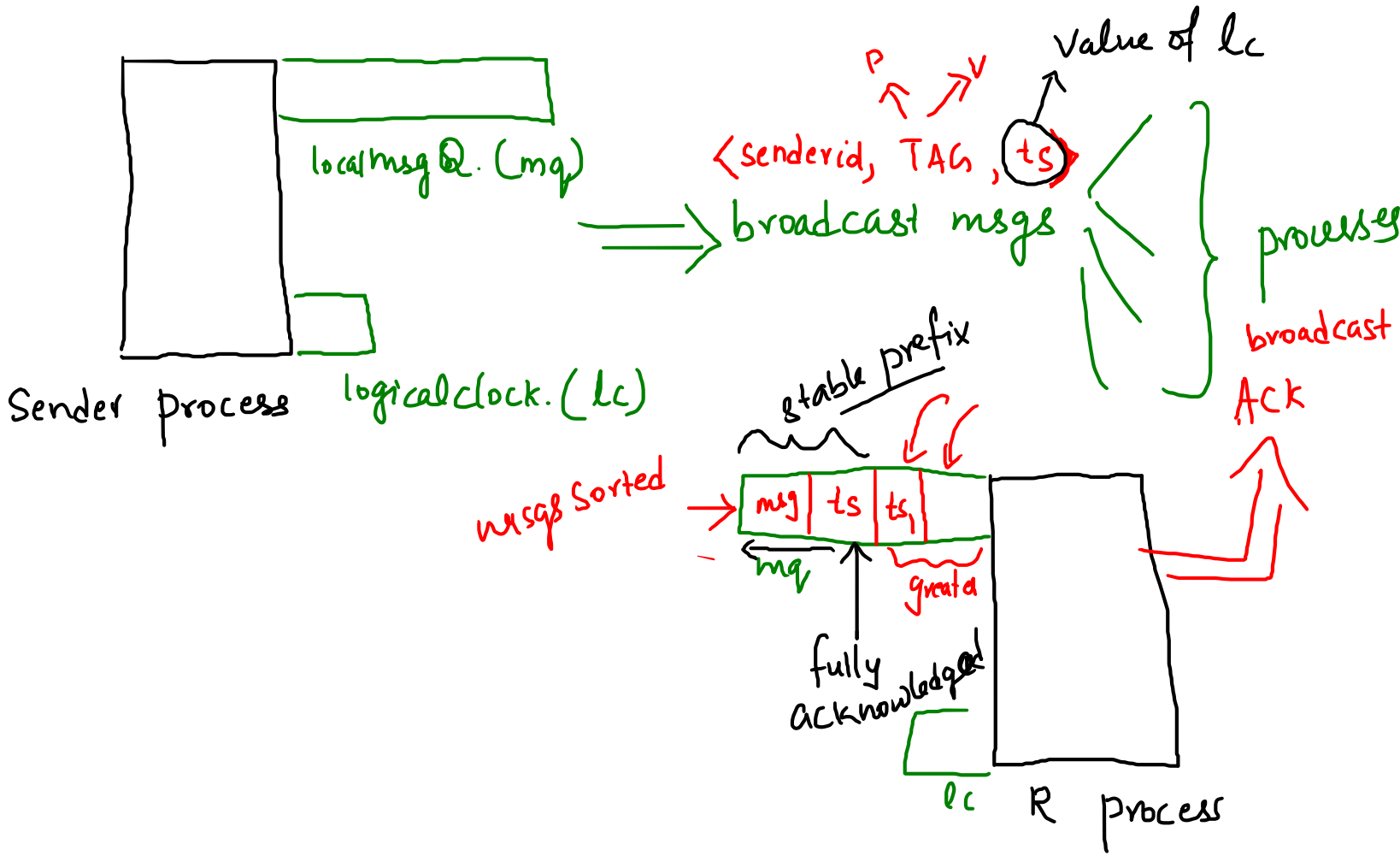
The no: of completed P ops is atmost the no: of completed V ops + the semaphore's initial value.

(1) how to represent the value of a semaphore

(2) and maintaining the semaphore invariant.

* Way to count P and V ops
* way to delay P ops

Broadcast msgs



local counters

nP , nV — # of fully acknowledged P & V oprns

execute P & V oprns stored in the stable prefix of mq

{ nV > nP
  increment nP } increment nV

— Delete the V or P msg from the mq

```
type kind = enum(V, P, ACK)
chan sem[1:n](sender: int, kind, timestamp: int)
chan go[1:n](timestamp : int)
User[i: 1..n]:: var lc : int := 0      # logical clock
                var ts : int           # timestamp in go messages
                # execute a V operation
                broadcast sem(i, V, lc); lc := lc+1
                ...
                # execute a P operation
                broadcast sem(i, P, lc); lc := lc+1
                receive go[i](ts); lc := max(lc, ts+1); lc := lc+1
                ...
Helper[i: 1..n]:: var mq : queue of (int, kind, int)   # ordered by timestamps
                  var lc : int := 0                    # logical clock
                  var nV : int := 0, nP : int := 0     # semaphore counters
                  var sender : int, k : kind, ts : int # values in messages
                  do true → {loop invariant DSEM}
                       receive sem[i](sender, k, ts); lc := max(lc, ts+1); lc := lc+1
                       if k = P or k = V →
                            insert (sender, k, ts) at appropriate place in mq
                            broadcast sem(i, ACK, lc); lc := lc+1
                       [] k = ACK →
                            record that another ACK has been seen
                            fa fully acknowledged V messages →
                                 remove the message from mq; nV := nV+1
                            af
                            fa fully acknowledged P messages st nV > nP →
                                 remove the message from mq; nP := nP+1
                                 if sender = i → send go[i](lc); lc := lc+1 fi
                            af
                       fi
                  od
```

Token Passing Algms

Token → Special kind of msg
- convey permission
- gather info^n

1. Critical Section Problem
2. Termination Detection in distributed computation

Token Passing Algms

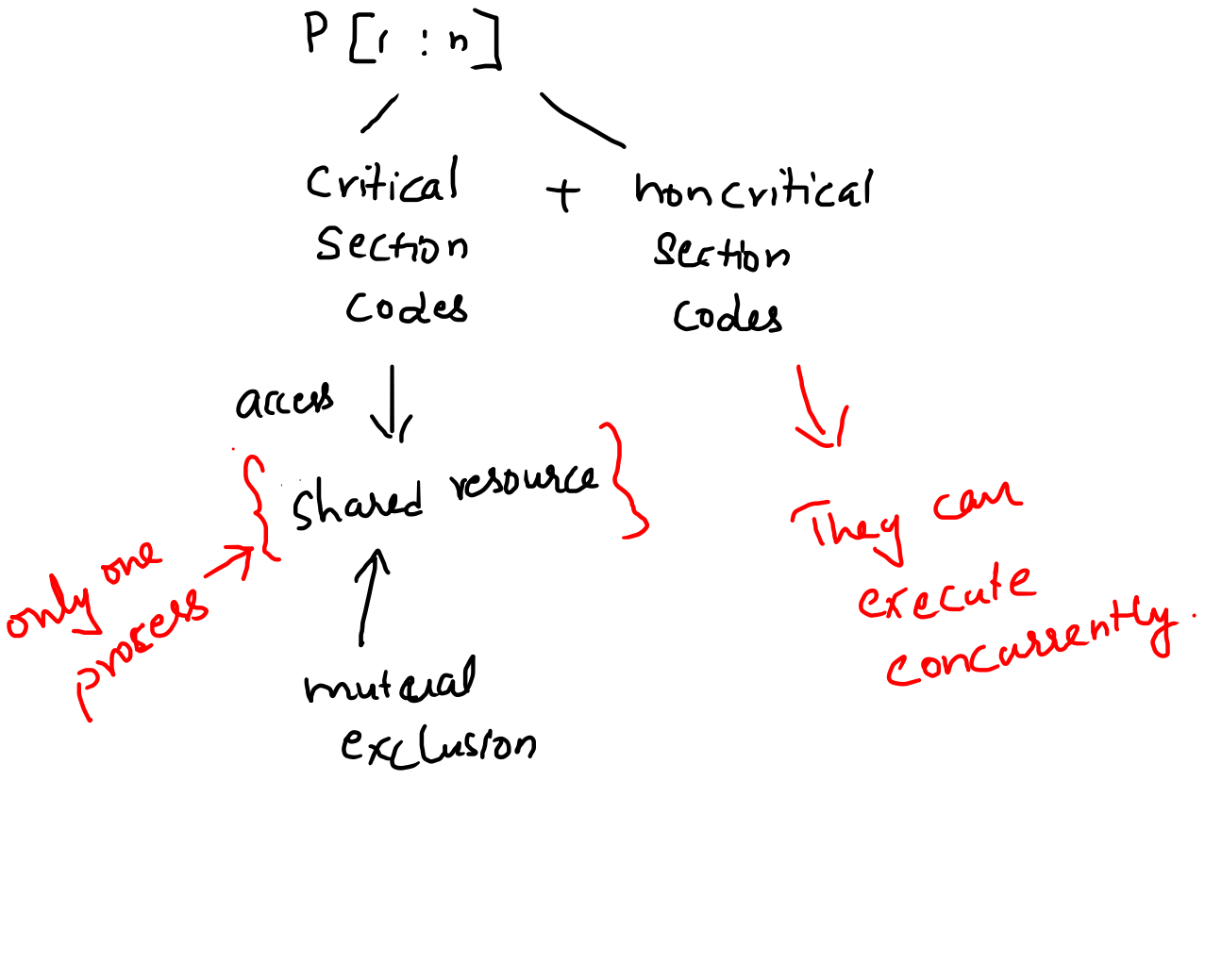
Token → Special kind of msg
- convey permission
- gather info^n

1. Critical Section Problem
2. Termination Detection in distributed computation

Distributed Mutual Exclusion

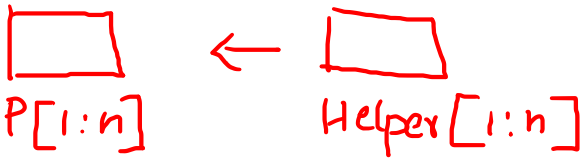
Critical Section problem :- It ensures that at most one process at a time executes code that accesses a shared resource

1. Active monitors → nonreplicated files
2. Distributed semaphores → decentralized solution
   "huge number of msgs to be exchanged"
3. Token ring → decentralized solution.

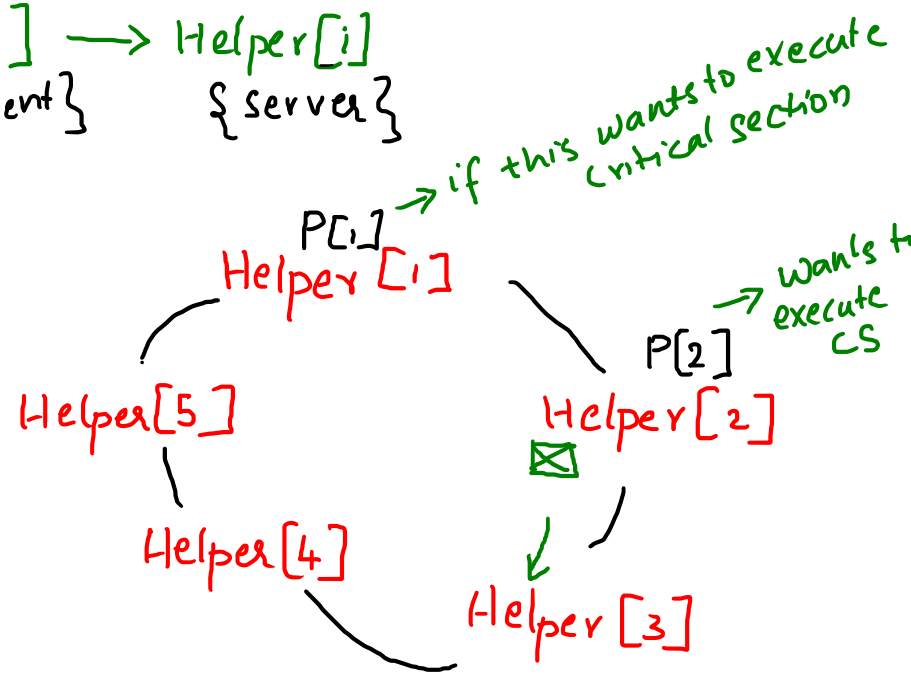


entry
exit
} protocols.

Tokens → to control the entry to critical section.



P[i] → Helper[i]
{client} {server}



Token is circulated b/w the Helpers.