Paradigms for Distributed Process Interaction

- Manager/Workers (Distributed Bag of Tasks)
- Heartbeat algorithms
- Pipeline Algorithms
- Probe/Echo Algorithms
- Broadcast Algorithms
- Token Passing
- Replicated Servers

Manager/Workers (Distributed Bag of Tasks)

- Central manager process implements bag, hands out tasks and collects results.
- Workers get tasks and deposit results.
- Essentially client-server.
- Easy to change number of workers (adapt to available resources).
- If many more tasks than workers, load is balanced effectively.

Example: Sparse Matrix Multiplication

Compute

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}$$

where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are sparse $N \times N$ matrices.

- Each row of $\mathbf{C}$ (= row of $\mathbf{A}$) is a task.
- First distribute $\mathbf{B}$ to all workers.
- Then distribute tasks to workers as the workers become available.

Manager

```
process Manager {
        for[ w : 0 to M-1 ] send worker[w]( B ) ;
        for[ w : 0 to M-1 ] send worker[w]( w, A[w] ) ;
        int done := 0;
        int nextRow := M ;
        while (done > M) {
                int w ; int i ; double R[N] ;
                receive manager( (w,i,R) ) ;
                C[i] := R ;
                if( nextRow < N ) {
                        send worker[w]( (nextRow, A[ nextRow ] )
                     )
                        nextRow += 1 ;
                else {
                        send worker[w]( -1, * ) ;
                        done += 1 ; } } }
```

(Could also use a rendezvous or monitor)

Worker

```
process Worker[ w : 0 to M-1 ] {
        double B[N][N] ;
        receive worker[w](B) ;
        while( true ) {
              int i ; double A[N], C[N] ;
              // Receive row i of matrix A.
              receive worker[w]( (i, A) ) ;
              if( i == -1 ) break ;
              for[ j : 0 to N-1 ] {
                    double c := 0.0 ;
                    for[ k : 0 to N-1 ] c += A[k] * B[k,j] ;
                    C[j] := c ; }
              send manager( (w,i,C) ) ; } }
```

Heartbeat Algorithms

- Peers cooperate by repeatedly exchanging information.
- Well suited to data-parallel iterative algorithms.
- Each process is responsible for a part of the data.
- New value of an element is dependent on values of neighbours.
- Each process sends to neighbours, then receives from neighbours.

```
process Worker[i = 1 to numWorkers] {
Initialize local variables
        while (! done) {
              send to neighbours
              receive from neighbours
              update local variables } }
```

Example: Region Labeling

- Assume binary image. (i.e., `image[m,n]` where each element (pixel) is 1 or 0.)
- *pixel neighbours* are above, below, left and right (not diagonals).
- We'd like to find `label[m,n]` s.t. if two neighbours are 1 then they have the same label
- and the number of label values is minimal.
- For SIMD machine, could use one processor per pixel.
- On distributed MIMD architecture it's best to divide image into strips or blocks.

Iterative Algorithm

```
for[ i : 0 to m-1] {
        for[ j : 0 to n-1 ] {
              // Put a unique label on each "white" pixel.
              if (image[i,j] == 1) label[i,j] := i*n+j; } }
do {
        change := false;
        for[ i : 0 to m-1] {
              for[ j : 0 to n-1 ] {
                    int oldlabel := label[i,j];
                    if (image[i,j] == 1) {
                          label[i,j] = max(label for neighbours);
                    }
                    if (label[i,j] != oldlabel)
                          change := true; }
} while( change ) ;
```

Heartbeat Algorithm

Each process has a horizontal strip of size “size” of the image + 1 row above + 1 row below.

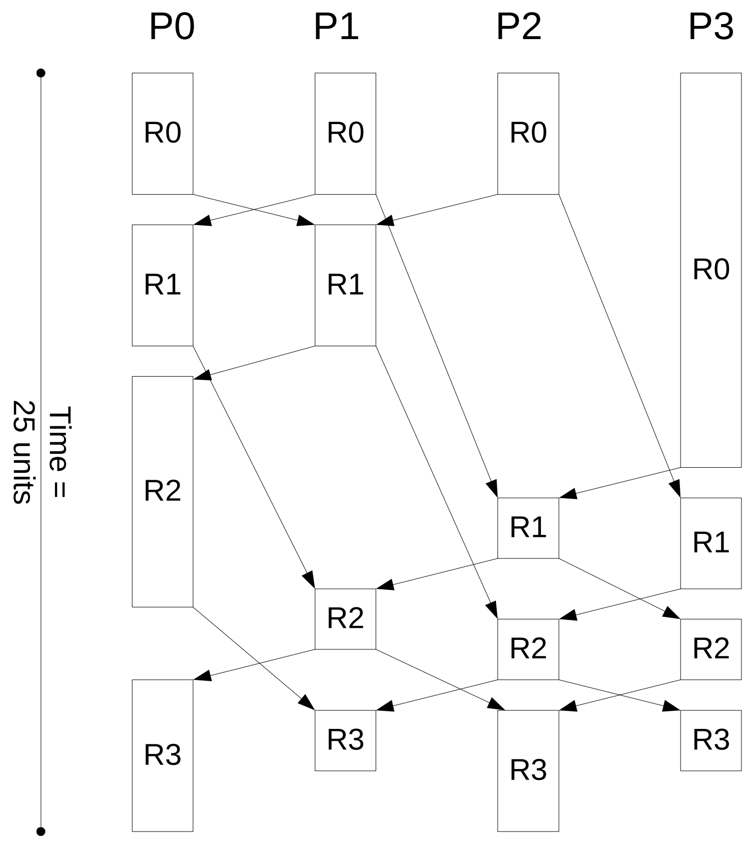
Each process is responsible for updating a corresponding strip of labels.

```
initialize the local strip of labels based only on local
information
do {
        send label[1,*] up (unless first worker)
        send label[size,*] down (unless last worker)
        receive label[0,*] (unless first worker)
        receive label[size+1,*] (unless last worker)
        bool change := false ;
        update labels for local strip, computing change.
        send change to coordinator
        receive change from coordinator
} while( change ) ;
```

Termination

- Iterative algorithm terminates when no label changes in an iteration.
- In distributed algorithm we need to detect when no label has changed globally – can't be done locally.
- Central coordinator works, but creates a bottleneck.
- Could use tree or other structure.

Heartbeat algorithms have local barriers



Heartbeat synchronization

In the example P0 starts round 2 while P3 is still in round 0. Total time is 25 units.Total time for full barrier synchronization is 33 units.

Pipeline Algorithms

- Each process receives from one process and sends to another
- Configurations

  Open: Endpoints are not connected.

  Closed: Coordinator inputs to first and collects results from last.

  Circular: Output from last is input to first.

Example: Pipeline Matrix Multiplication

Coordinator (pipeline is closed)

1. send rows of $\mathbf{A}$ to first worker
2. send columns of $\mathbf{B}$ to first worker
3. receive rows of $\mathbf{C}$, in reverse order, from last worker

Worker

1. receive my row of $\mathbf{A}$ from upstream
2. receive other rows of $\mathbf{A}$ from upstream and send them downstream.
3. receive columns of $\mathbf{B}$ from upstream and send them downstream.
4. Compute inner product.
5. Send my row of $\mathbf{C}$ downstream.
6. Receive other rows of $\mathbf{C}$ from upstream and send them downstream.

Matrix Multiplication by Blocks (2D pipeline)

- Arrange processes in grid: process (i, j) calculates $C[i, j]$.
- A values circulate leftward. B values circulate upward.
- In iteration k, process (i, j) multiplies $A[i, i \oplus j \oplus k]$ by $B[i \oplus j \oplus k, j]$, ($\oplus$ is addition mod n)
- Process (i, j) initially has $A[i, i \oplus j]$ and $B[i \oplus j, j]$

A[0,0] B[0,0]	A[0,1] B[1,1]	A[0,2] B[2,2]
A[1,1] B[1,0]	A[1,2] B[2,1]	A[1,0] B[0,2]
A[2,2] B[2,0]	A[2,0] B[0,1]	A[2,1] B[1,2]

```
## a = A[i,i⊕j] and b = B[i⊕j,j]
c := a * b;
for [k = 1 to N-1] {
        send a left; send b up;
        receive a from right; receive b from below;
        ## a = A[i,i⊕j⊕k] and b = B[i⊕j⊕k,j]
        c += a * b; }
```

Probe/Echo Algorithms

- Concurrent analog to depth-first-search.
- Distributed computation modeled as a graph.
  - Process = Node
  - Channel = Edge
- Processes communicate only with their neighbours (assume bidirectional).

Broadcast — Probe Algorithm

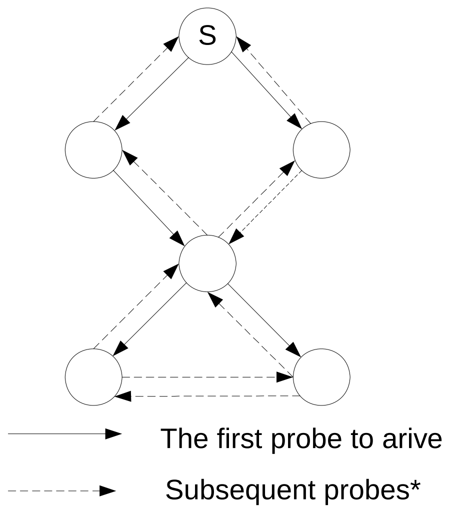
- Node `S` wants to send a message `m` to all other nodes in a graph.
- **Spanning Tree**:
  - ∗ Assume `S` knows a spanning tree, `t` rooted at `S`.
  - ∗ Send `m` and `t` to each child in `t`.
  - ∗ Each node forwards `m` only to children in `t`.
  - ∗ `m` received once *and only once* by each other process.
- **Neighbour-set**:
  - ∗ Each node only knows its own neighbours.
  - ∗ `S` sends `m` to all its neighbours.
  - ∗ When node receives `m` it forwards it to *all* of its neighbours (including originator).
  - ∗ Every nodes receives from and sends to every neighbor.
  - ∗ Thus each node can expect to receive the same message $\deg(n)$ times.

Discover Topology — Probe/Echo Algorithm

- Initially each node only knows about its neighbours.
- If network is an undirected tree
  - $*$ S sends probe to all neighbors.
  - $*$ Other nodes, upon receiving probe send probe to all other neighbors.
  - $*$ Leaf replies (echo) with it's neighbour-set filled in.
  - $*$ When node has received echo for each probe, it merges neighbor-sets and replies to its parent.
  - $*$ When S receives echo from all its children, then it knows the full topology.
- For general connected graph:
  - $*$ S sends probe to all neighbors.
  - $*$ On first probe received: probe is resent to all neighbours.
  - $*$ On subsequent probe: echo with null graph $(\varnothing, \varnothing)$.
  - $*$ Number of echoes expected at each node = number of probes sent.
  - $*$ Once all echoes received: union of all information known and (except S) reply to first prober

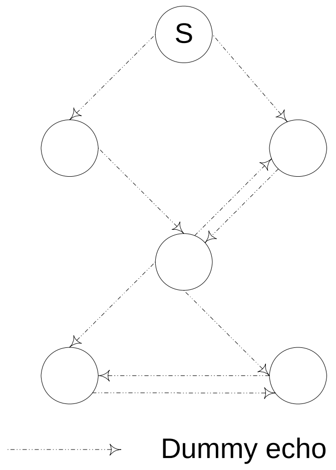
Node *S* wants to learn the topology.

- *S* sends probes to all its neighbours
- When a node other than *S* receives its first probe it sends probes to all its neighbours.
- Thus the first pobes to arrive form a tree and each node "knows" what its parent is.
- But nodes do not yet "know" which their children are and the leaves do not "know" they are leaves.
- Every node, *n*, will receive deg(*n*) probes.

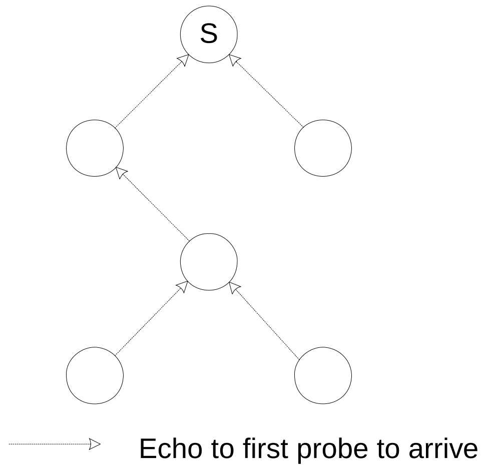


* The original sender, *S*, considers all probes to be subsequent.

- When a node receives a subsequent probe it immediately sends back a "dummy echo" containing no useful information.
- A leaf, *n*, will get deg(*n*) dummy echoes



- Once a node, *n*, has received deg(*n*) echoes, it echoes its parent with useful information collected from itself and its children.
- Note the order of these echoes is bottom up.



Broadcast Algorithms

Many networks support a `broadcast` primitive. We assume:

- Each process receives one copy of a broadcast message (including originator).
- Broadcasts are received on the same channel(s) as point-to-point messages.
- Broadcasts from the same process will be received in the order that they were sent.
- Broadcast is not atomic – processes may receive messages in different order.

Logical Clocks

Consider each process as a sequence of events including the sending and receiving of messages.

If every thread had a perfectly synchronized real-time clock we could assign a total order on events.

But such synchronization is not feasible in a distributed system.

Events: a, b, c, ...

Consider an ordering relation $a \longrightarrow b$ meaning a happened before b. Define it as the smallest relation such that

- If a and b are in the same thread and a precedes b in that thread, then $a \to b$.
- If a is a sending of a message and b is the receiving of the same message then $a \to b$.
- If $a \to b$ and $b \to c$ then $a \to c$

A logical clock is a mapping C from events to integers such that

- For all a and b, if $a \to b$ then $C(a) < C(b)$

A logical clock provides an order in which events "could have happened"

We can think of the global state of the system at time i as consisting of the states of the processes and channels after all events a, with $C(a) < i$, have happened.

Implementing a logical clock

We can implement a logical clock as follows.

Each process has a local "logical clock" variable lc.

lc is incremented after each event occurs. For internal events

```
$a$                  ⇝    $a$
                          $C(a)$ := lc ;
                          lc += 1 ;
```

We append a "timestamp" to each message.

A send event a is implements as

```
$a$ : send ch(m) ;   ⇝    send ch(m,lc) ;
                          $C(a)$ := lc ;
                          lc += 1 ;
```

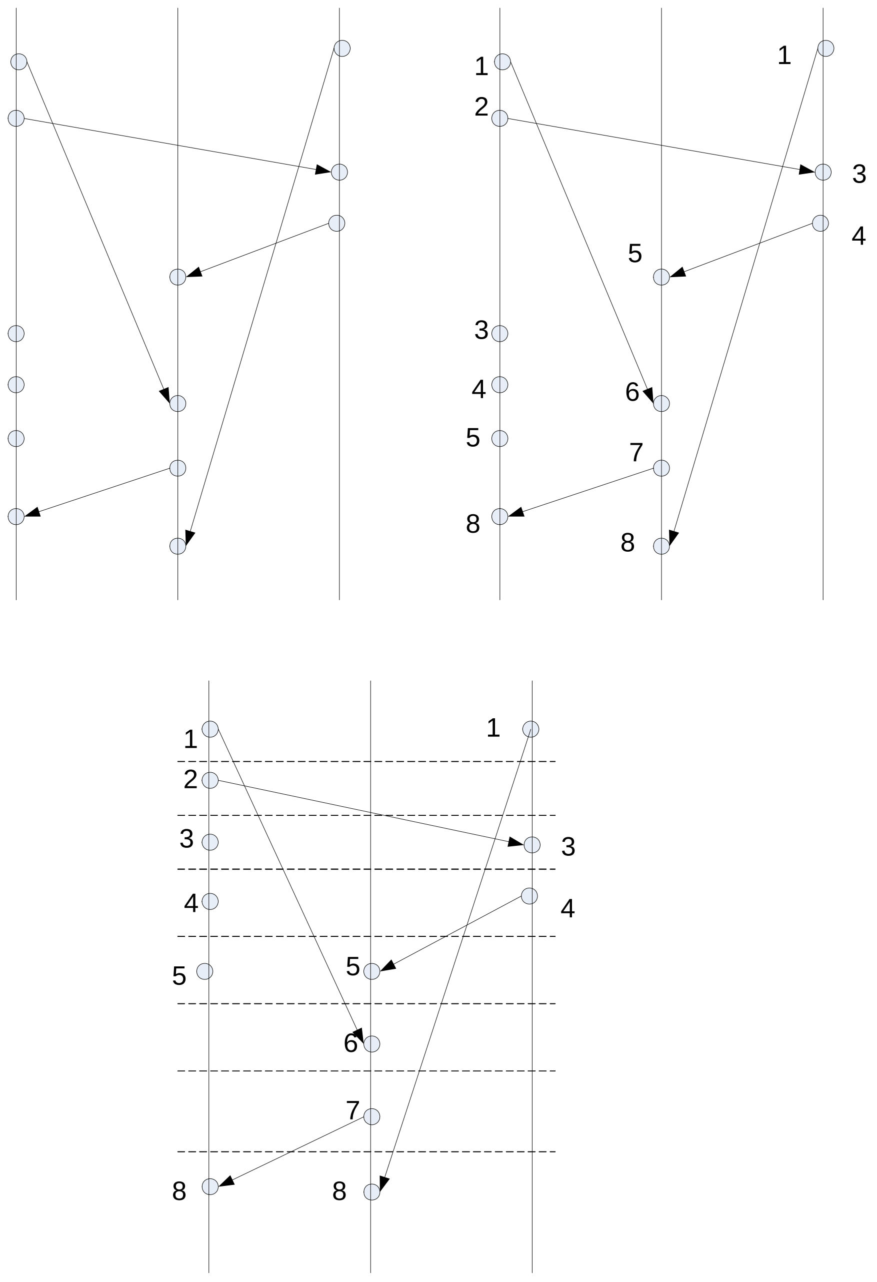
And a receive event a is implemented by

```
receive ch(m) ;
```

$\leadsto$

```
int ts ;
receive ch(m,ts) ;
lc := max( lc, ts+1 ) ;
$C(a) :=$ lc ;
lc += 1 ;
```

Logical Clocks Example



Breaking ties

Note that $C(a) \leq C(b)$ is only a partial order on events as $C(a) = C(b)$ is possible even if $a \neq b$.

In some applications we need a total order. We can use the process number to break ties. I.e. if $P(a)$ is the number of the process that a occurs in then

$$C(a) < C(b) \vee (C(a) = C(b) \wedge P(a) < P(b))$$

is a total order.

Using timestamps to ensure information is up to date.

If we assume that all send events on a given channel are only from one process.

- if process p has received a message on channel c with a time stamp of t no future message on c will have a time stamp of $< t$
- So if p has received a message from all its incoming channels with a time stamp of t or more then no future message will have a time of $< t$

Example: Distributed Semaphores

Each process p_i communicates only with a handler process h_i.

- p_i sends h_i VREQ and PREQ messages.
- h_i sends p_i GO messages.
- V operation is : send VREQ to h_i
- P operation is : send PREQ to h_i ; receive GO from h_i.

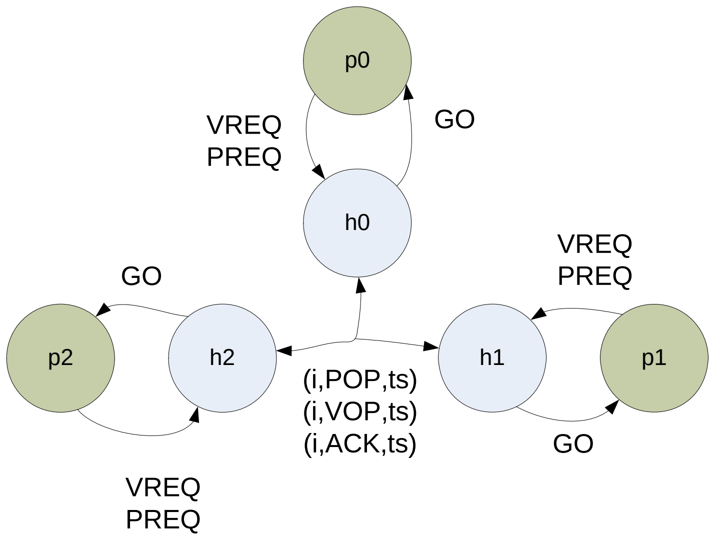
Distributed processes implement semaphore-style synchronization without a central coordinator.

- The handler h_i communicate with each other with broadcasts of
  - $*$ (i, POP, ts) — when it gets a PREQ
  - $*$ (i, VOP, ts) — when it gets a VREQ
  - $*$ (i, ACK, ts) — when it gets a POP or VOP.
- Each handler maintains:
  - $*$ a logical clock, `lc`,
  - $*$ a queue, `mq`, of received POP or VOP messages, sorted by msg timestamp (sender's pid used to break ties), and
  - $*$ a local int value for the semaphore, `s`.
  - $*$ (For all c) When a process has some received message

from all other processes with `ts` $> c$, then all messages in `mq` with `ts` $\le c$ are *fully acknowledged*.

- Fully acknowledged messages in `mq` form a *stable prefix* — this process can never see a message that will be inserted in that part.
- When POP or VOP message is received, broadcast an acknowledgment — used by other processes to ensure that operation is fully acknowledged.
- When ACK is received, update stable prefix:
  - $*$ for VOP — increment s, delete msg.
  - $*$ for POP — if `s` > 0, decrement `s`, delete msg.
    - $\cdot$ If the POP was from i send a GO message.
- Doesn't scale well:
  - $*$ every process is involved in every synchronization,
  - $*$ n^2 messages per P or V,
  - $*$ doesn't tolerate communications failures.

Example



Suppose there are three processes $\{p_0, p_1, p_2\}$ and three helpers $\{h_0, h_1, h_2\}$.

Let's consider h_0. Initially $s = 0, lc = 0, mq = \langle\rangle$.

- Receive PREQ from p_0
- Broadcast $(0, POP, 0)$, $lc := 1$
- Receive $(0, POP, 0), lc := 2, mq := \langle(0, POP, 0)\rangle$
- Broadcast $(0, ACK, 2)$, $lc := 3$
- Receive $(0, ACK, 2), lc = 4$
- Receive $(1, POP, 0)$, $lc = 5, mq := \langle(0, POP, 0) , (1, POP, 0)\rangle$

- Broadcast $(0, ACK, 5)\,, lc := 6$
- Receive $(1, ACK, 4)\,, lc := 7$
- Receive $(2, ACK, 3)\,, lc := 8$

At this point both messages in mq are fully acknowledged. However, as $s = 0$, both messages are blocked.

- Receive
  $$(2, VOP, 6)\,, lc := 9,$$
  $$mq := \left\langle \underline{(0, POP, 0)\,, (1, POP, 0)}, (2, VOP, 6) \right\rangle$$
- Broadcast $(0, ACK, 9)\,, lc := 10$
- Receive $(1, ACK, 8)\,, lc := 11$

At this point all messages in mq are fully acknowledged.

We repeatedly process the stable prefix from left to right, skipping blocked messages, until all messages in the stable prefix are blocked.

- $mq := \left\langle \underline{(0, POP, 0)\,, (1, POP, 0)} \right\rangle, s := 1$
- $mq := \left\langle \underline{(1, POP, 0)} \right\rangle, s := 0$
- Send (GO) to p_0

Token-Passing Algorithms

Token — a special message used to convey permission or gather global state information.

Distributed Mutual Exclusion

`Helper` processes

- connected in a (logical) ring
- continuously pass the token around the ring
- when user requests `CS`
  - wait for the token
  - hold token while user in `CS`

"On-demand" Distributed Mutual Exclusion Method

`Helper` processes

- when user i requests `CS`
  - helper i sends request counterclockwise
  - when the request reaches the token holding process, it waits until it is not in `CS` then sends the token clockwise
  - when the token reaches a requesting process it is held

Termination Detection

How to determine if a distributed computation is terminated?

Assuming one-way ring structure (all messages on ring, all work results from messages, messages do not cross in channel):

- There is one token.
- A process is considered idle when it is waiting for a message or terminated.
- When a process:
  - ∗ becomes idle and has the token, it passes the token;
  - ∗ is active and receives the token, it keeps the token;
  - ∗ is idle and receives the token, it passes the token;
  - ∗ receives the token and has been *continuously* idle since it last sent it, then computation is done:
    - · no messages are in transit
    - · every other process must be idle
- Once one process knows the computation is over, it can broadcast this fact.

What about if network isn't a ring?

- Construct a cycle, `C`, that includes every edge in the network (`nc` = length of `C`).
- Add a counter to the token.
- Processes have "colour": blue = idle, red = active, initially red.

On receiving a regular message:

```
colour := red
```

On receiving the token

```
if (token >= nc) halt
else {
    if (colour == red) {
        colour := blue;
        token := 0; }
    else
        token++;
    send token along the next edge in C }
```

The token makes one circuit to turn all processes blue and another to verify they are still all blue

Replicated Servers

Multiple processes, each acting as a server.

- To manage distinct instances of a resource, or
- to give illusion of a single resource.

Dining Philosopher's revisited

Consider adding `Waiter`(s) to manage the forks

Centralized: One waiter controls all the forks.

Distributed: One waiter per fork. (Requires asymmetry or preemption to avoid deadlock.)

Decentralized: One waiter per philosopher. Adjacent waiters share 1 fork.

Hygienic Philosophers (Decentralized)

- Five philosophers, five waiters, five forks
  - $p_i \longrightarrow w_i$: *hungry* (when ready to eat)
  - $w_i \longrightarrow p_i$: *eat* (when waiter has 2 forks)
  - $p_i \longrightarrow w_i$: *release* (when finished eating)
- Each fork is a token, held by one of two adjacent waiters (or in transit).
  - $w_i \longrightarrow w_{i+1}$: *needL* (when waiter i needs its right fork)
  - $w_{i+1} \longrightarrow w_i$: *passR* (passes i its right fork)
  - $w_i \longrightarrow w_{i-1}$: *needR* (when waiter i needs its left fork)
  - $w_{i-1} \longrightarrow w_i$: *passL* (passes i its left fork)
- Need to avoid livelock (ping-ponging)
  - After waiter acquires a fork, it won't give it up until the philosopher has used it at least once.
  - Implementation: Each fork is marked dirty or clean
    - Forks are clean when they arrive.
    - Forks become dirty when the philosopher eats.
    - Only dirty forks are passed

- Example: (○ clean, ● dirty)
  - ∗ Initial state (from 0 to 4) $(\bullet|\bullet)(|\bullet)(|\bullet)(|\bullet)(|)$
    - · $p_4 \longrightarrow w_4 : \mathit{hungry}$.
    - · $w_4 \longrightarrow w_0 : \mathit{needL}$.
    - · $w_4 \longrightarrow w_3 : \mathit{needR}$.
    - · $w_0 \longrightarrow w_4 : \mathit{passR}$. Now 4 has 1 fork. $(|\bullet)(|\bullet)(|\bullet)(|\bullet)(|\circ)$
    - · $p_0 \longrightarrow w_0 : \mathit{hungry}$.
    - · $w_0 \longrightarrow w_4 : \mathit{needR}$. Since the fork is clean, w_4 won't give it up.
    - · $w_3 \longrightarrow w_4 : \mathit{passL}$. Now 4 has 2 forks. $(|\bullet)(|\bullet)(|\bullet)(|)(\circ|\circ)$
    - · $w_4 \longrightarrow p_4 : \mathit{eat}$. Philosopher eats.
    - · $p_4 \longrightarrow w_4 : \mathit{release}$. $(|\bullet)(|\bullet)(|\bullet)(|)(\bullet|\bullet)$
    - · $w_4 \longrightarrow w_0 : \mathit{passL}$. Now request can be honoured. $(\circ|\bullet)(|\bullet)(|\bullet)(|)(\bullet|)$

Deadlock

- Since a clean fork can't be passed and both forks must be at the same waiter to become dirty, the state $(|\circ)(|\circ)(|\circ)(|\circ)(|\circ)$ is deadlocked.
  - $*$ But from state $(\bullet|\bullet)(|\bullet)(|\bullet)(|\bullet)(|)$ one can't reach a state $(|\circ)(|\circ)(|\circ)(|\circ)(|\circ)$ nor $(\circ|)(\circ|)(\circ|)(\circ|)(\circ|)$
  - $*$ Challenge: Figure out why.
- Freedom from starvation.
  - $*$ Once the fork is dirty, the waiter is generous with it.
  - $*$ Priority is to other philosopher.
  - $*$ Thus a philosopher waits only until its neighbors have finished eating.

Decentralized File Server

One process has the file (token). Complete connection graph of servers.

When a server wants access to a file, it asks for the token.

The token will only be passed when the client is finished.

Priority goes to the server that had the token least recently.

Can be extended to allow read-only copies.

- If a file is open for reading, then a copy of the file (but not the token) can be passed immediately.
- If the requesting server already has a valid copy, then that can be used.
- When the file is later opened for writing, all copies must first be invalidated.

(Similar system can be used for cache coherency)

Object Replication

Extreme case

- All servers holding replica are involved in each write.
- Any server can be used for reading

General case. $W\dot{Q} > N/2$ is the write quota. $RQ > N - WQ$ is the read quota.

- N servers hold replicas
- At least WQ servers are involved in each write operation.
- Each write is time stamped (logical clocks)
- At least RQ servers are required for first read.
- The value with the latest time stamp wins.
- One advantage is that the system still works when $N - WQ$ servers are down.

Object Replication (Example)

$N = 10, WQ = 7, RQ = 4$.

- Initially, the value is 42 with timestamp of 0. All servers have a copy.
- Server 0 wants to write a value of 13. It must recruit 6 other servers.
  - Server 0 recruits servers 1 thru 6 (since $WQ = 7$)
  - All 7 change the value to 13 with time stamp 1.
- Server 9 wants to read. It must recruit 3 other servers.
  - It recruits servers $6, 7, 8$
  - Servers $7, 8, 9$ reports the value is 42, but 6 reports it is 13.
  - The value 13 wins because of its time stamp.

Object Replication — Deadlock

As you can see, recruitment could deadlock if one is not careful.

Consider: $N = 10, WQ = 7$.

- If server 0 wants to write and recruits 1 thru 4
- and server 9 wants to write are recruits 5 thru 8,
- then we are deadlocked, unless there is some way for a server to unrecruit itself.
  - (which creates many more complications)
- Simple solution: Always recruit in ascending order.