

Paradigms for Process Interaction

Andrews, Chapter 09

Three basic patterns for distributed programs

- producer/consumer
- client/server
- interacting peers

Basic patterns can be combined in several paradigms

- Parallel computation:
 - Manager/workers
 - * distributed bag of tasks
 - Heartbeat algorithms
 - * periodically send then receive
 - Pipeline algorithms
 - * information flows with receive then send
- Distributed systems:
 - Probes (sends) and echoes (receives)
 - * disseminate then gather in trees and graphs
 - Broadcast algorithms
 - * decentralized decision making
 - Token-passing algorithms
 - * another approach to decentralized decision making
 - Replicated server processes
 - * manage multiple instances of resources

§9.1 Manager/Workers (Distributed Bag of Tasks)

§9.1.1 Sparse matrix representation

- Compute product $A \times B = C$ of $n \times n$ matrices.
- Requires n^2 inner products.
- A matrix is *dense* if most entries are nonzero.
- A matrix is *sparse* if most entries are zero.
- Sparse matrix representation:

```
int lengthA[n];  
pair *elementsA[n]
```

- Example:

lengthA	elementsA
1	(3, 2.5)
0	
0	
2	(1, -1.5) (4, 0.6)
0	
1	(0, 3.4)

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 & 2.5 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.5 & 0.0 & 0.0 & 0.6 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 3.4 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

Sparse matrix multiplication

- Represent A and C by rows, B by columns.
- Each row of A is a task.
- Each task will need all columns of B .

```

module Manager
  type pair = (int index, double value);
  op getTask(result int row, len; result pair [*]elems);
  op putResult(int row, len; pair [*]elems);
body Manager
  int lengthA[n], lengthC[n];
  pair *elementsA[n], *elementsC[n];
  # matrix A is assumed to be initialized
  int nextRow = 0, tasksDone = 0;

  process manager {
    while (nextRow < n or tasksDone < n) {
      # more tasks to do or more results needed
      in getTask(row, len, elems) ->
        row = nextRow;
        len = lengthA[i];
        copy pairs in *elementsA[i] to elems;
        nextRow++;
      [] putResult(row, len, elems) ->
        lengthC[row] = len;
        copy pairs in elems to *elementsC[row];
        tasksDone++;
    }
  }
end Manager

```

Figure 9.1 (a) Sparse matrix multiplication: Manager process.

```

process worker[w = 1 to numWorkers] {
    int lengthB[n];
    pair *elementsB[n]; # assumed to be initialized
    int row, lengthA, lengthC;
    pair *elementsA, *elementsC;
    int r, c, na, nb; # used in computing
    double sum; # inner products
    while (true) {
        # get a row of A, then compute a row of C
        call getTask(row, lengthA, elementsA);
        lengthC = 0;
        for [i = 0 to n-1]
            INNER_PRODUCT(i); # see body of text
        send putResult(row, lengthC, elementsC);
    }
}

```

Figure 9.1 (b) Sparse matrix multiplication: Worker processes.


```

sum = 0.0; na = 1; nb = 1;
c = elementsA[na]->index;      # column in row of A
r = elementsB[i][nb]->index;   # row in column of B
while (na <= lengthA and nb <= lengthB) {
  if (r == c) {
    sum += elementsA[na]->value *
           elementsB[i][nb]->value;
    na++; nb++;
    c = elementsA[na]->index;
    r = elementsB[i][nb]->index;
  } else if (r < c) {
    nb++; r = elementsB[i][nb]->index;
  } else { # r > c
    na++; c = elementsA[na]->index;
  }
}
if (sum != 0.0) { # extend row of C
  elementsC[lengthC] = pair(i, sum);
  lengthC++;
}

```

Inner product code for Worker *i* in sparse matrix multiplication.

§9.1.2 Adaptive Quadrature Revisited

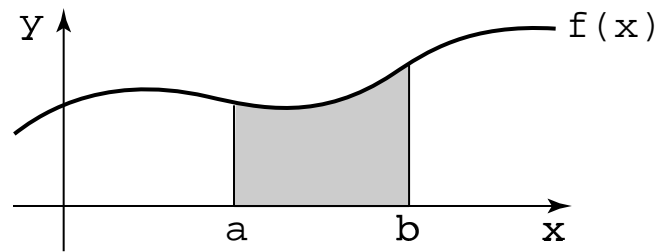


Figure 1.4 The quadrature problem.

Copyright © 2000 by Addison Wesley Longman, Inc.

```
double fleft = f(a), fright, area = 0.0;
double width = (b-a) / INTERVALS;
for [x = (a + width) to b by width] {
    fright = f(x);
    area = area + (fleft + fright) * width / 2;
    fleft = fright;
}
```

Iterative Quadrature Program

Copyright © 2000 by Addison Wesley Longman, Inc.

```

double quad(double left,right,fleft,fright,lrarea) {
    double mid = (left + right) / 2;
    double fmid = f(mid);
    double larea = (fleft+fmid) * (mid-left) / 2;
    double rarea = (fmid+fright) * (right-mid) / 2;
    if (abs((larea+rarea) - lrarea) > EPSILON) {
        # recurse to integrate both halves
        larea = quad(left, mid, fleft, fmid, larea);
        rarea = quad(mid, right, fmid, fright, rarea);
    }
    return (larea + rarea);
}

```

Recursive Procedure for Quadrature Problem

Copyright © 2000 by Addison Wesley Longman, Inc.

```

module Manager
  op getTask(result double left, right);
  op putResult(double area);
body Manager
  process manager {
    double a, b;          # interval to integrate
    int numIntervals;     # number of intervals to use
    double width = (b-a)/numIntervals;
    double x = a, totalArea = 0.0;
    int tasksDone = 0;
    while (tasksDone < numIntervals) {
      in getTask(left, right) st x < b ->
        left = x; x += width; right = x;
      [] putResult(area) ->
        totalArea += area;
        tasksDone++;
    }
    print the result totalArea;
  }
end Manager

double f() { ... }          # function to integrate
double quad(...) { ... }   # adaptive quad function

process worker[w = 1 to numWorkers] {
  double left, right, area = 0.0;
  double fleft, fright, lrarea;
  while (true) {
    call getTask(left, right);
    fleft = f(left); fright = f(right);
    lrarea = (fleft + fright) * (right - left) / 2;
    # calculate area recursively as shown in Section 1.5
    area = quad(left, right, fleft, fright, lrarea);
    send putResult(area);
  }
}

```

- Combination of iterative and recursive

Figure 9.2 Adaptive quadrature using manager/workers paradigm.

§9.2 Heartbeat Algorithms

```
process Worker[i = 1 to numWorkers] {  
    declarations of local variables;  
    initialize local variables;  
    while (not done) {  
        send values to neighbors;  
        receive values from neighbors;  
        update local values;  
    }  
}
```

Structure of heartbeat algorithms.

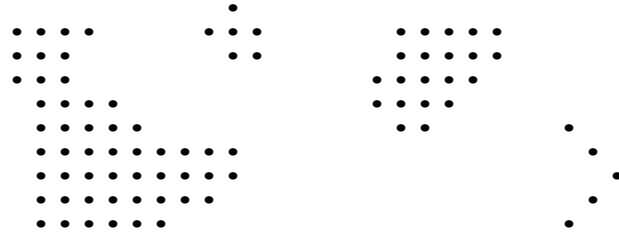
Copyright © 2000 by Addison Wesley Longman, Inc.

- Useful for data parallel iterative applications.
- Each worker updates part of the data.
- Each worker then requires data from its neighbors to continue.

Heartbeat algorithms

- If the data is a grid it can be broken up into strips or blocks.
- 3D data can be broken up into planes, prisms, or cubes.
- Examples:
 - region labelling
 - Game of Life

§9.2.1 Image Processing: Region Labeling



Sample image for the region-labeling problem.

Copyright © 2000 by Addison Wesley Longman, Inc.

- Image has *three* regions (using 4-neighbors).
- Initially each point is given a unique label: $mi + j$.
- Iterate until no changes:
 - If adjacent pixels are lit, set label of each to maximum.
- Final labels are regions.

Region labelling

- We could assign a task to each pixel.
- This would be appropriate on a SIMD machine, such as a graphics card.
- For MIMD machines, these tasks are too small.
- Break image up into strips.
- On each iteration, each task needs to exchange its edge values with its neighbor.
- Each individual task could examine its pixels once, or (to cut down on the messaging) iterate until they don't change.
- Workers cannot determine when to terminate.
- Coordinator detects termination by receiving messages from all workers.
- Could speed up termination detection with a butterfly.

```

chan first[1:P](int edge[n]);      # for exchanging edges
chan second[1:P](int edge[n]);
chan answer[1:P](bool);           # for termination check

process Worker[w = 1 to P] {
    int stripSize = m/W;
    int image[stripSize+2,n];      # local values plus edges
    int label[stripSize+2,n];      # from neighbors
    int change = true;
    initialize image[1:stripSize,*] and label[1:stripSize,*];

    # exchange edges of image with neighbors
    if (w != 1)
        send first[w-1](image[1,*]);      # to worker above
    if (w != P)
        send second[w+1](image[stripSize,*]); # to below
    if (w != P)
        receive first[w](image[stripSize+1,*]); # from below
    if (w != 1)
        receive second[w](image[0,*]);     # from worker above

    while (change) {
        exchange edges of label with neighbors, as above;
        update label[1:stripSize,*] and set change to true if
            the value of the label changes;
        send result(change);               # tell coordinator
        receive answer[w](change);         # and get back answer
    }
}

```

Figure 9.3 (a) Region labeling: Worker processes.

```

chan result(bool);  # for results from workers

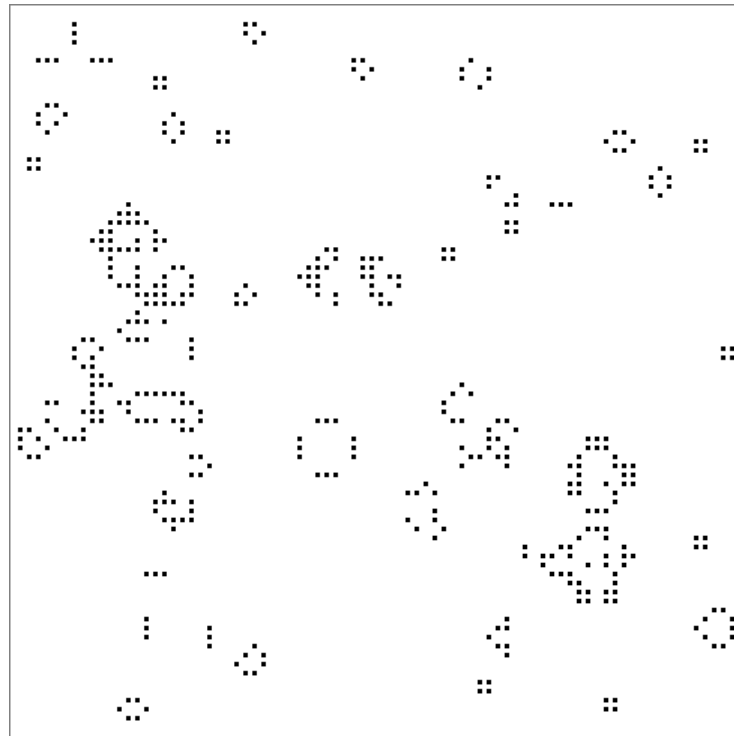
process Coordinator {
    bool chg, change = true;
    while (change) {
        change = false;
        # see if there has been a change in any strip
        for [i = 1 to P] {
            receive result(chg);
            change = change or chg;
        }
        # broadcast answer to every worker
        for [i = 1 to P]
            send answer[i](change);
    }
}

```

Figure 9.3 (b) Region labeling: Coordinator process.

Game of Life

- <http://pmav.eu/stuff/javascript-game-of-life-v3.1.1/>
- I have provided a Racket implementation (from rosettacode.org).
- Rules:
 - A live cell with zero or one live neighbors dies from loneliness.
 - A live cell with two or three live neighbors survives.
 - A live cell with four or more live neighbors dies from overpopulation.
 - A dead cell with exactly three live neighbors becomes alive.



```

chan exchange[1:n,1:n](int row, column, state);

process cell[i = 1 to n, j = 1 to n] {
    int state;    # initialize to dead or alive
    declarations of other variables;
    for [k = 1 to numGenerations] {
        # exchange state with 8 neighbors
        for [p = i-1 to i+1, q = j-1 to j+1]
            if (p != q) (p != i or q != j)
                send exchange[p,q](i, j, state);
        for [p = 1 to 8] {
            receive exchange[i,j](row, column, value);
            save value of neighbor's state;
        }
        update local state using rules in text;
    }
}

```

Figure 9.4 The Game of Life.

§9.3 Pipeline Algorithms

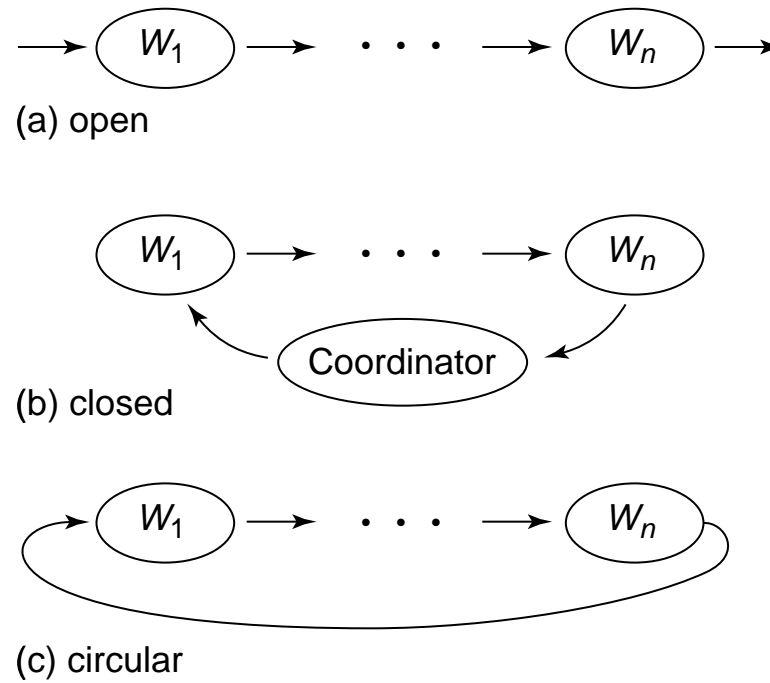


Figure 9.5 Pipeline structures for parallel computing.

§1.8 used a circular pipeline for distributed matrix multiplication.

```
process worker[i = 0 to n-1] {
    double a[n];          # row i of matrix a
    double b[n];          # one column of matrix b
    double c[n];          # row i of matrix c
    double sum = 0.0;      # storage for inner products
    int nextCol = i;      # next column of results
    receive row i of matrix a and column i of matrix b;
    # compute  $c[i,i] = a[i,*] \times b[*,i]$ 
    for [k = 0 to n-1]
        sum = sum + a[k] * b[k];
    c[nextCol] = sum;
    # circulate columns and compute rest of  $c[i,*]$ 
    for [j = 1 to n-1] {
        send my column of b to the next worker;
        receive a new column of b from the previous worker;
        sum = 0.0;
        for [k = 0 to n-1]
            sum = sum + a[k] * b[k];
        if (nextCol == 0)
            nextCol = n-1;
        else
            nextCol = nextCol-1;
        c[nextCol] = sum;
    }
    send result vector c to coordinator process;
}
```



```

chan vector[n](double v[n]);  # messages to workers
chan result(double v[n]);     # rows of c to coordinator

process Coordinator {
    double a[n,n], b[n,n], c[n,n];
    initialize a and b;
    for [i = 0 to n-1]          # send all rows of a
        send vector[0](a[i,*]);
    for [i = 0 to n-1]          # send all columns of b
        send vector[0](b[:,i]);
    for [i = n-1 to 0]          # receive rows of c
        receive result(c[i,*]); # in reverse order
}

```

Figure 9.6 (a) Matrix multiplication pipeline: Coordinator process.

```

process Worker[w = 0 to n-1] {
    double a[n], b[n], c[n]; # my row or column of each
    double temp[n];          # used to pass vectors on
    double total;            # used to compute inner product

    # receive rows of a; keep first and pass others on
    receive vector[w](a);
    for [i = w+1 to n-1] {
        receive vector[w](temp); send vector[w+1](temp);
    }

    # get columns and compute inner products
    for [j = 0 to n-1] {
        receive vector[w](b); # get a column of b
        if (w < n-1)         # if not last worker, pass it on
            send vector[w+1](b);
        total = 0.0;
        for [k = 0 to n-1]    # compute one inner product
            total += a[k] * b[k];
        c[j] = total;        # put total into c
    }

    # send my row of c to next worker or coordinator
    if (w < n-1)
        send vector[w+1](c);
    else
        send result(c);
    # receive and pass on earlier rows of c
    for [i = 0 to w-1] {
        receive vector[w](temp);
        if (w < n-1)
            send vector[w+1](temp);
        else
            send result(temp);
    }
}

```

Figure 9.6 (b) Matrix multiplication pipeline: Worker processes.

Properties of the circular solution

- Messages chase each other around the pipeline: rows of a , then columns of b , then rows of c .
- There is little delay between receiving a message and passing it along.
- Once a has been distributed, inner products with b are computed very fast.
- The number of columns of b is arbitrary.
- Each worker could have a strip of rows of a .
- Pipeline could be open and part of a larger pipeline.

Matrix multiplication by row/column circular queues.

$a_{1,2}, b_{2,1}$	$a_{1,3}, b_{3,2}$	$a_{1,4}, b_{4,3}$	$a_{1,1}, b_{1,4}$
$a_{2,3}, b_{3,1}$	$a_{2,4}, b_{4,2}$	$a_{2,1}, b_{1,3}$	$a_{2,2}, b_{2,4}$
$a_{3,4}, b_{4,1}$	$a_{3,1}, b_{1,2}$	$a_{3,2}, b_{2,3}$	$a_{3,3}, b_{3,4}$
$a_{4,1}, b_{1,1}$	$a_{4,2}, b_{2,2}$	$a_{4,3}, b_{3,3}$	$a_{4,4}, b_{4,4}$

Initial arrangement for matrix multiplication by blocks.

Copyright © 2000 by Addison Wesley Longman, Inc.

- Use n^2 processes and $2n^2$ channels.
- Shift rows and columns initially as above.
- On each iteration, shift a's to the left, b's up.
- After $n - 1$ shifts, each process has computed its inner product.
- Can use blocks instead of single cells to reduce number of processes and channels.

```

chan left[1:n,1:n](double);  # for circulating a left
chan up[1:n,1:n](double);    # for circulating b up

process Worker[i = 1 to n, j = 1 to n] {
    double aij, bij, cij;
    int LEFT1, UP1, LEFTI, UPJ;
    initialize above values;

    # shift values in aij circularly left i columns
    send left[i,LEFTI](aij); receive left[i,j](aij);
    # shift values in bij circularly up j rows
    send up[UPJ,j](bij); receive up[i,j](bij);
    cij = aij * bij;

    for [k = 1 to n-1] {
        # shift aij left 1, bij up 1, then multiply and add
        send left[i,LEFT1](aij); receive left[i,j](aij);
        send up[UP1,j](bij); receive up[i,j](bij);
        cij = cij + aij*bij;
    }
}

```

Figure 9.7 Matrix multiplication by blocks.

§9.4 Probe/Echo Algorithms

- Distributed version of depth-first search.

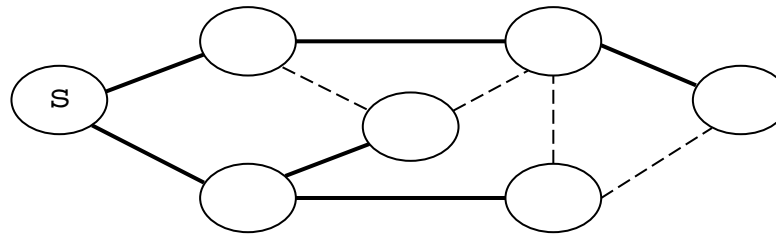


Figure 9.8 A spanning tree of a network of nodes.

Copyright © 2000 by Addison Wesley Longman, Inc.

- If a single node knows the network topology:
 - compute the spanning tree
 - send message and tree to all descendents in tree
 - each descendent forwards message and tree to its descendents
- Network topology:
 - data structure describing all connections of all nodes

```

type graph = bool [n,n];
chan probe[n](graph spanningTree; message m);

process Node[p = 0 to n-1] {
    graph t; message m;
    receive probe[p](t, m);
    for [q = 0 to n-1 st q is a child of p in t]
        send probe[q](t, m);
}

process Initiator { # executed on source node s
    graph topology = network topology;
    graph t = spanning tree of topology;
    message m = message to broadcast;
    send probe[s](t, m);
}

```

Figure 9.9 Network broadcast using a spanning tree.

What if we don't know the topology?

- How can a node know how many times to receive?
- Too few: extra messages are left in the message queues.
- Too many: deadlock waiting for messages never sent.

```

chan probe[n](message m);

process Node[p = 1 to n] {
    bool links[n] = neighbors of node p;
    int num = number of neighbors;
    message m;
    receive probe[p](m);
    # send m to all neighbors
    for [q = 0 to n-1 st links[q]]
        send probe[q](m);
    # receive num-1 redundant copies of m
    for [q = 1 to num-1]
        receive probe[p](m);
}

process Initiator { # executed on source node s
    message m = message to broadcast;
    send probe[s](m);
}

```

Figure 9.10 Broadcast using neighbor sets.

Copyright © 2000 by Addison Wesley Longman, Inc.

- Topology not necessary.
- Each node knows exactly how many messages to receive.
- Advanced topic: fault-tolerant broadcast.

Computing the topology of a network

- Two phases:
 - probe: as before, each node sends to all others
 - echo: each node returns local topology
- When the start node receives all echoes, it has the global topology.
- Could then be efficiently broadcast to all nodes.

Computing the topology of an acyclic network (tree)

- Initiator (root) sends probe to all children.
- Each node receives probe from parent, sends probe on to children.
- Leaf nodes receive probe from parent, echo their local topology.
- After an internal node receives all echoes, sends echo to parent including local topology.
- When the root receives all echoes, it will have global topology.

```

type graph = bool [n,n];
chan probe[n](int sender);
chan echo[n](graph topology)    # parts of the topology
chan finalecho(graph topology)  # final topology

process Node[p = 0 to n-1] {
    bool links[n] = neighbors of node p;
    graph newtop, localtop = ([n*n] false);
    int parent;    # node from whom probe is received
    localtop[p,0:n-1] = links;    # initially my links

    receive probe[p](parent);
    # send probe to other neighbors, who are p's children
    for [q = 0 to n-1 st (links[q] and q != parent)]
        send probe[q](p);

    # receive echoes and union them into localtop
    for [q = 0 to n-1 st (links[q] and q != parent)] {
        receive echo[p](newtop);
        localtop = localtop or newtop;    # logical or
    }
    if (p == S)
        send finalecho(localtop);    # node S is root
    else
        send echo[parent](localtop);
}

process Initiator {
    graph topology;
    send probe[S](S)    # start probe at local node
    receive finalecho(topology);
}

```

Figure 9.11 Probe/echo algorithm for gathering the topology of a tree.

Computing the topology of a general network (graph)

- After receiving a probe, each node sends probes to all its neighbors.
- Each node may thus receive multiple probes.
- All but the first are echoed immediately with `null` topology.
- Eventually a node will receive echoes from every probe.
- It keeps the union of all these echoes, adding them to its local topology.
- After receiving an echo from every node, the node sends an echo to the *first* probing node with the accumulated topology.

```

type graph = bool [n,n];
type kind = (PROBE, ECHO);
chan probe_echo[n](kind k; int sender; graph topology);
chan finalecho(graph topology);

process Node[p = 0 to n-1] {
    bool links[n] = neighbors of node p;
    graph newtop, localtop = ([n*n] false);
    int first, sender; kind k;
    int need_echo = number of neighbors - 1;
    localtop[p,0:n-1] = links;    # initially my links

    receive probe_echo[p](k, first, newtop); # get probe
    # send probe on to all other neighbors
    for [q = 0 to n-1 st (links[q] and q != first)]
        send probe_echo[q](PROBE, p, ∅);

    while (need_echo > 0) {
        # receive echoes or redundant probes from neighbors
        receive probe_echo[p](k, sender, newtop);
        if (k == PROBE)
            send probe_echo[sender](ECHO, p, ∅);
        else # k == ECHO {
            localtop = localtop or newtop; # logical or
            need_echo = need_echo-1;
        }
    }
    if (p == S)
        send finalecho(localtop);
    else
        send probe_echo[first](ECHO, p, localtop);
}

process Initiator {
    graph topology; # network topology
    send probe_echo[source](PROBE, source, ∅);
    receive finalecho(topology);
}

```

- Unified channel: can receive probes or echoes at any time.

Figure 9.12 Probe/echo algorithm for computing the topology of a graph.

§9.5 Broadcast Algorithms

- In previous section we considered networks connected in a graph.
- In local area networks, processors share a common communication channel.
- In this situation, it is easy to support **broadcast** messages, which transmit a message from one process to all the others.

```
broadcast ch(m);
```

```
co [i=1 to n]  
  send ch[i](m);
```

- Processes use `receive` for both kinds of messages.
- broadcast is not atomic:
 - broadcast messages from A and B could arrive in any order.

§9.5.1 Logical clocks and event ordering

- Actions of processes are either local or communication actions.
- Communication actions must be synchronized.
- In this section, *event* refers to execution of send, broadcast, or receive.

§9.5.1 Logical clocks and event ordering

- There exists a partial ordering of events:
 - sending a message must *happen before* the receiving of the same message.
- This *happens before* relation is reflexive, antisymmetric, and transitive: a **partial order**.
- Not every pair of events is in the ordering:
 - if A sends a message to B and then C, the arrivals of these messages are not ordered.

§9.5.1 Logical clocks and event ordering

- If we had a global clock, we could impose a total ordering with timestamps.
- But perfect synchronization of local clocks is impossible.
- A **logical clock** is an integer counter that is incremented when events occur.

Logical clock update rules.

Let A be a process and let lc be a logical clock in the process.

A updates the value of lc as follows:

1. When A sends or broadcasts a message, it sets the timestamp of the message to the current value of lc and then increments lc by 1.
2. When A receives a message with timestamp ts , it sets lc to the maximum of lc and $ts + 1$ and then increments lc by 1.

Clock values and a total order for events using logical clocks

- Every send event the clock value is the timestamp of the message.
- Every receive event the clock value is the maximum of lc and $ts + 1$ (but before incrementing).
- These rules ensure that every event has a clock value.
- These rules also ensure that if an event a *happens before* another event b , the clock value of a will be smaller than the clock value of b .
- We break ties (same clock value) by smaller process ID to get a total order.

Distributed Semaphores

- We could use semaphores in a distributed environment by implementing them on a server.
- We can also use semaphores in a distributed environment by decentralizing them.

Distributed Semaphores

- Semaphore is an integer s
- Invariant:
 - Number of successful P operations is less or equal to number of V operations plus initial value of s .
 - To implement, we need a way to count P and V operations, and delay P operations.
- Invariant: $s \geq 0$
 - Processes which share a semaphore need to maintain this.

Distributed Semaphores

- Processes broadcast when they want to P or V:
 - message includes ID, timestamp, and POP or VOP.
- Processes keep POP and VOP messages in a queue m_q , sorted by timestamp.
- Processes also keep their own POP and VOP messages in this queue.
- If all messages were received in order, every process would know all the P and V commands and could maintain the invariants.
- Unfortunately, broadcast is not atomic.
 - Messages broadcast by two different processes can be received in different orders by different processes.

Distributed Semaphores

- However, consecutive messages sent by each process *do* have increasing timestamps.
- Therefore:
 - Suppose a process's message queue mq contains a message m with timestamp ts .
 - Once the process has received a message with a larger timestamp from every other process, it knows it will *never* see a message with a smaller timestamp.
 - When this happens the message m is said to be **fully acknowledged**.
- Further, if m is fully acknowledged, then so are all messages in front of it in the queue.
- Therefore, the part of the queue up to and including m is a **stable prefix**:
 - no new messages will ever be inserted into it.

ACK messages

- If some process never sends a POP or VOP, nothing will ever be fully acknowledged.
- Possibility of deadlock. Therefore:
- After each process *receives* a POP or VOP message, it will broadcast an *ACK* message.
- ACK messages have timestamps and update the logical clocks, but are not stored in the message queue mq.
- Thus they facilitate the full acknowledgement of other messages.

Distributed semaphore implementation

- Each process maintains its own local integer variable s .
- For every VOP message, increment s and delete the message from mq .
- Examine POP messages in stable prefix in timestamp order:
 - if $s > 0$ decrement s and delete the POP message.
- Invariant *DSEM*:
$$s \geq 0 \wedge mq \text{ is ordered by timestamps}$$
- POP messages are processed in stable prefix order.
- All processes handle POP messages in same order.

```

type kind = enum(reqP, reqV, VOP, POP, ACK);
chan semop[n](int sender; kind k; int timestamp);
chan go[n](int timestamp);

process User[i = 0 to n-1] {
    int lc = 0, ts;
    ...
    # ask my helper to do V(s)
    send semop[i](i, reqV, lc); lc = lc+1;
    ...
    # ask my helper to do P(s), then wait for permission
    send semop[i](i, reqP, lc); lc = lc+1;
    receive go[i](ts); lc = max(lc, ts+1); lc = lc+1;
}

process Helper[i = 0 to n-1] {
    queue mq = new queue(int, kind, int); # message queue
    int lc = 0, s = 0; # logical clock and semaphore
    int sender, ts; kind k; # values in received messages
    while (true) { # loop invariant DSEM
        receive semop[i](sender, k, ts);
        lc = max(lc, ts+1); lc = lc+1;
        if (k == reqP)
            { broadcast semop(i, POP, lc); lc = lc+1; }
        else if (k == reqV)
            { broadcast semop(i, VOP, lc); lc = lc+1; }
        else if (k == POP or k == VOP) {
            insert (sender, k, ts) at appropriate place in mq;
            broadcast semop(i, ACK, lc); lc = lc+1;
        }
        else { # k == ACK
            record that another ACK has been seen;
            for (all fully acknowledged VOP messages in mq)
                { remove the message from mq; s = s+1; }
            for (all fully acknowledged POP messages in mq st s > 0) {
                remove the message from mq; s = s-1;
                if (sender == i) # my user's P request
                    { send go[i](lc); lc = lc+1; }
            }
        }
    }
}

```

Figure 9.13 Distributed semaphores using a broadcast algorithm.

Distributed Semaphores

- We can use distributed semaphores in distributed systems the same way we did in shared memory systems.
 - mutual exclusion
 - barriers
 - *etc.*
- Broadcast messages and logical clocks can be used to solve other problems as well.
- Every process takes part in every decision, so it does not scale well to large numbers of processes.
- In addition, it must be modified to be fault tolerant.

§9.6 Token-Passing Algorithms

§9.6.1 Distributed mutual exclusion

- Critical sections primarily arise in shared memory programs.
- Often distributed programs must manage a resource that can only be used by a single process at a time:
 - communication link to a satellite
 - distributed file system or database
- Best solution is often an active monitor.
- Another solution is distributed semaphores.
 - no one process has a centralized role
 - but all processes share all decisions
 - lots of broadcast and ACK messages
- **Token ring** is a third solution.
 - decentralized and fair
 - requires far fewer messages than distributed semaphores

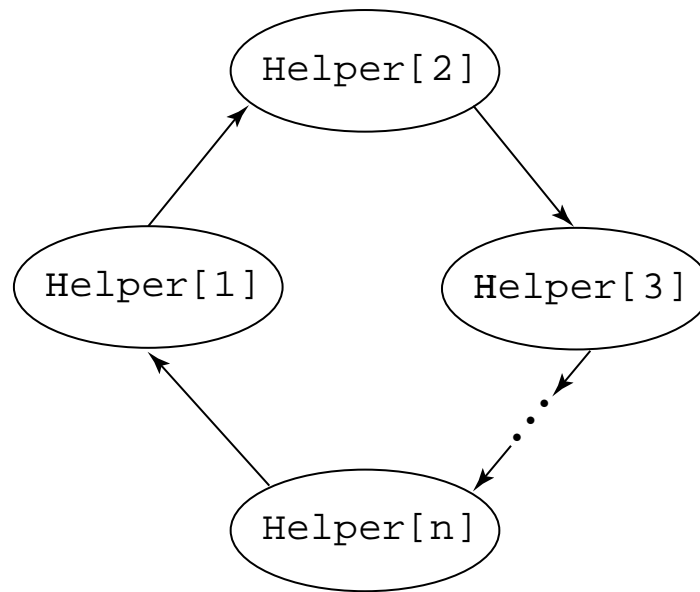


Figure 9.14 A token ring of helper processes.

Copyright © 2000 by Addison Wesley Longman, Inc.

- *DMUTEX:*

User[i] in CS \Rightarrow Helper[i] has token
 \wedge
there is exactly one token

```

chan token[1:n](), enter[1:n](), go[1:n](), exit[1:n]();

process Helper[i = 1 to n] {
  while (true) {    # loop invariant DMUTEX
    receive token[i]();          # wait for token
    if (not empty(enter[i])) {   # does user want in?
      receive enter[i]();        # accept enter msg
      send go[i]();              # give permission
      receive exit[i]();         # wait for exit
    }
    send token[i%n + 1]();       # pass token on
  }
}

process User[i = 1 to n] {
  while (true) {
    send enter[i]();             # entry protocol
    receive go[i]();
    critical section;
    send exit[i]();              # exit protocol
    non-critical section;
  }
}

```

Figure 9.15 Mutual exclusion with a token ring.

§9.6.2 Termination detection in a ring

Global invariant *RING*:

$$T[1] \text{ is blue} \Rightarrow (T[1] \dots T[\text{token}+1] \text{ are blue} \wedge \\ \text{ch}[2] \dots \text{ch}[\text{token}\%n + 1] \text{ are empty})$$

actions of $T[1]$ when it first becomes idle:

```
color[1] = blue; token = 0; send ch[2](token);
```

actions of $T[2], \dots, T[n]$ upon receiving a regular message:

```
color[i] = red;
```

actions of $T[2], \dots, T[n]$ upon receiving the token:

```
color[i] = blue; token++; send ch[i%n + 1](token);
```

actions of $T[1]$ upon receiving the token:

```
if (color[1] == blue)
    announce termination and halt;
color[1] = blue; token = 0; send ch[2](token);
```

Figure 9.16 Termination detection in a ring.

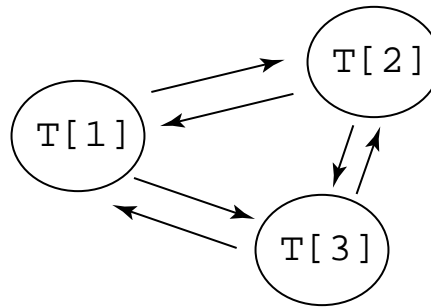


Figure 9.17 A complete communication graph.

Copyright © 2000 by Addison Wesley Longman, Inc.

Global invariant *GRAPH*:

token has value $V \Rightarrow$
(the last V channels in cycle **C** were empty \wedge
the last V processes to receive the token were **blue**)

actions of **T[i]** upon receiving a regular message:

color[i] = red;

actions of **T[i]** upon receiving the token:

if (token == nc)

 announce termination and halt;

if (color[i] == red)

 { **color[i] = blue; token = 0; }**

else

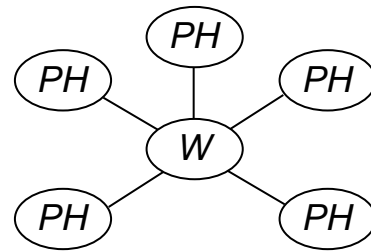
token++;

set **j** to index of channel for next edge in cycle **C**;

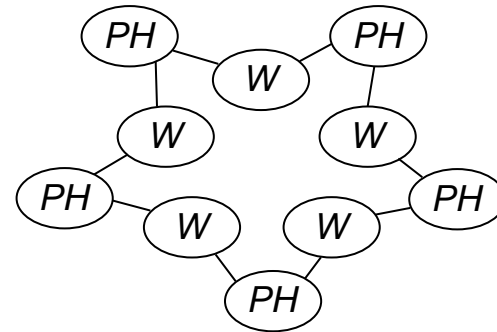
send ch[j](token);

Figure 9.18 Termination detection in a complete graph.

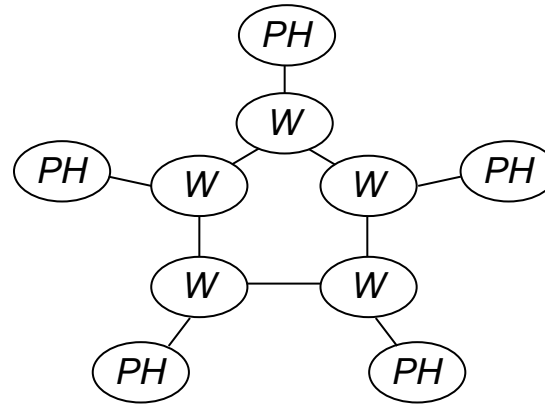
§9.7 Replicated Servers



(a) Centralized



(b) Distributed



(c) Decentralized

Figure 9.19 Solution structures for the dining philosophers.

```

module Waiter[5]
  op getforks(), relforks();
body
  process the_waiter {
    while (true) {
      receive getforks();
      receive relforks();
    }
  }
end Waiter

process Philosopher[i = 0 to 4] {
  int first = i, second = i+1;
  if (i == 4) {
    first = 0; second = 4; }
  while (true) {
    call Waiter[first].getforks();
    call Waiter[second].getforks();
    eat;
    send Waiter[first].relforks();
    send Waiter[second].relforks();
    think;
  }
}

```

Figure 9.20 Distributed dining philosophers.

```

module Waiter[t = 0 to 4]
  op getforks(int), relforks(int); # for philosophers
  op needL(), needR(),           # for waiters
    passL(), passR();
  op forks(bool,bool,bool,bool); # for initialization
body
  op hungry(), eat();           # local operations
  bool haveL, dirtyL, haveR, dirtyR; # status of forks
  int left = (t-1) % 5;         # left neighbor
  int right = (t+1) % 5;        # right neighbor

  proc getforks() {
    send hungry(); # tell waiter philosopher is hungry
    receive eat(); # wait for permission to eat
  }

  process the_waiter {
    receive forks(haveL, dirtyL, haveR, dirtyR);
    while (true) {
      in hungry() ->
        # ask for forks I don't have
        if (!haveR) send Waiter[right].needL();
        if (!haveL) send Waiter[left].needR();
        # wait until I have both forks
        while (!haveL or !haveR)
          in passR() ->
            haveR = true; dirtyR = false;
          [] passL() ->
            haveL = true; dirtyL = false;
          [] needR() st dirtyR ->
            haveR = false; dirtyR = false;
            send Waiter[right].passL();
            send Waiter[right].needL()
          [] needL() st dirtyL ->
            haveL = false; dirtyL = false;
            send Waiter[left].passR();
            send Waiter[left].needR();

          ni
        # let philosopher eat, then wait for release
        send eat(); dirtyL = true; dirtyR = true;
        receive relforks();
      [] needR() ->
        # neighbor needs my right fork (its left)
        haveR = false; dirtyR = false;
        send Waiter[right].passL();
      [] needL() ->
        # neighbor needs my left fork (its right)
        haveL = false; dirtyL = false;
        send Waiter[left].passR();

      ni
    }
  }
}
end Waiter

```



```

process Philosopher[i = 0 to 4] {
    while (true) {
        call Waiter[i].getforks();
        eat;
        call Waiter[i].relforks();
        think;
    }
}

process Main { # initialize the forks held by waiters
    send Waiter[0].forks(true, true, true, true);
    send Waiter[1].forks(false, false, true, true);
    send Waiter[2].forks(false, false, true, true);
    send Waiter[3].forks(false, false, true, true);
    send Waiter[4].forks(false, false, false, false);
}

```

Figure 9.21 Decentralized dining philosophers.