Contents:

I. NOT INCLUDED

II. Resource Allocation
   [9.] Burns Philosophers
   [9A.] Distributed Philosophers

III. NOT INCLUDED

IV. Leader Election
   [12.] LeLann-Chang-Roberts
   [13.] Peterson
   [13A.] Flood

V. Common Knowledge

VI. Global Snapshots

VII. Consensus/Byzantine Generals

VIII. Spanning Trees
   [20.] Gallager-Humblet-Spira MST
   [21.] Breadth-First Search
   [21A.] Bellman-Ford Shortest Path
   [21B.] Asynchronous SP

IX. Logical Clocks
   [22.] Lamport’s Partial Order
   [23.] Lamport’s Total Order for Mutual Exclusion

X. Termination Detection: Dijsktra-Scholten

XI. Phased Algorithms
   [24.] BFS and Leader Election
II. Resource Allocation
Given: $n$ processes and $m$ resources.
Problem: The mutual exclusion problem is a special case of the resource allocation problem: $n$ processes contend for a set of $m$ resources. An example of the resource allocation problem is the Dining Philosophers Problem where $n$ philosophers have access to $n$ resources and each philosopher requires mutually exclusive access to two of the resources at a time.
There are $n$ philosophers seated around a table. Each philosopher is either thinking (R), hungry (T), eating (C), or just finished eating (E). In order to eat, each philosopher needs two forks; $n$ forks are placed on the table such that there is one fork on the left and one to the right of each philosopher, see Figure 1. Each philosopher can pick up forks located immediately to his left or right only when the neighbor, with whom the fork is shared, does not have the fork.
As in mutual exclusion problems, there are two other desired conditions: progress and fairness. In the case of the philosophers, progress means that some philosophers are changing regions and fairness means that all philosophers get to eat eventually (no starvation).

Figure 1: $n = 4$ Dining Philosophers ($p_i$) and $n = 4$ forks ($F_i$)
Algorithm: Burns’ Dining Philosophers

Mutual Exclusion: YES Progress: YES Fairness: YES Single-Writer: NO

Outline: Each resource (fork) is associated with a semaphore. Even numbered processes pick up
their left forks first and odd numbered processes pick their right forks first.

Shared Variables:

1. $FORK$: semaphore array $[0..n - 1]$, initially all available, where $FORK_i$ is written and read
   by $p_{i-1}$ and $p_i$

Code: At $p_i \in \{0,...,n - 1\}$:

   do forever
     if even($i$) then {
       wait($FORK_{i+1}$) /* left fork */
       wait($FORK_i$) /* right fork */
     }
     if odd($i$) then {
       wait($FORK_i$) /* right fork */
       wait($FORK_{i+1}$) /* left fork */
     }
   /* Critical region */
   signal($FORK_i$)
   signal($FORK_{i+1}$)
   /* Remainder region */
9A. Algorithm: Distributed Dining Philosophers
Consider a message-passing version of $n$ dining philosopher processes with an additional process which keeps all $m$ semaphores locally. That is, the philosophers are distributed but the semaphores, with FIFO queues, are centralized within the “keeper” process. A “wait” on a semaphore is a send to the keeper and a receive of an acknowledgement. The keeper checks the status of the specified semaphore and, if it is available, an acknowledge is sent out immediately. If it is not available, then the sender’s port is queued on that particular semaphore. A “signal” on a semaphore is just a send to the keeper, which then examines the status of that semaphore. If there any waiters, a port is dequeued and the delayed acknowledgement is sent.

**Code:** At philosopher $p_i \in \{1, \ldots, n\}$:

```cpp
do forever
    wait(0)  /* or as many as needed of the $m$ semaphores */
    wait(1)
    /* critical section */
    signal(0)
    signal(1)
```

**Local Variables:**

1. $counts$: array $[0..m - 1]$ of integer, initially all $1$
2. $queues$: array $[0..m - 1]$ of FIFO queues, initially all empty

**Code:** At semaphore keeper $p_i = n + 1$:

```cpp
do forever
    message ← receive()
    if message is a WAIT then
        if $- counts[sem] < 0$ then
            enqueue the sender’s port
        else
            send acknowledgement to sender’s port
    if message is a SIGNAL then
        if $counts[sem] + + < 0$ then {
            dequeue a sender’s port
            send acknowledgement to sender’s port
        }
```
IV. Leader Election

Given: \( n \) processes computing on a network configured in a ring architecture, see Figure 2. Each process has a unique identifier, which may be different from the process IDs.

**Problem:** Elect a *leader* from among the processes based on the identifier. For example, the leader may be designated to regenerate a *token* or, in general architectures, to be a *root* in a spanning tree. It is important that all processes know the identifier of the leader.

![Diagram of a ring network with processes at each node forwarding messages](image)

*Figure 2: Ring Network with Processes at each Node Forwarding Messages*
Algorithm: LeLann-Chang-Roberts’ Leader Election

Outline: Each process sends its own ID to the next counterclockwise neighbor on the ring. A process receives a message with an ID from its next clockwise neighbor and compares the ID to its own ID. The algorithm sends messages in only one direction (unidirectional) and does not require that the value of $n$ be known. The synchronous time complexity is $n$ rounds for the maximum ID to circle the ring and the message complexity is $O(n^2)$.

- If the received ID is larger, it is sent on to the next node.
- If the received ID is smaller, it is ignored.
- If the received ID is equal, the process is the leader since its ID traveled all the way around the ring.

Local Variables:

1. $temp\_id \leftarrow i$, the temporary ID of the process
2. $next\_temp\_id$, the temporary ID of the next clockwise process
3. $neighbor \leftarrow i + 1$, the next counterclockwise process on the ring

Code: At $p_i \in \{1, \ldots, n\}$:

send($neighbor$, $temp\_id$)

do forever

$next\_temp\_id \leftarrow$ receive()

if $next\_temp\_id > temp\_id$ then send($neighbor$, $next\_temp\_id$)
if $next\_temp\_id = temp\_id$ then announce($LEADER$)
Algorithm: Peterson’s Leader Election

Outline: The processes elect the process which is currently “holding” the maximum identifier as a temporary ID. Each process sends its own temporary ID to the next counterclockwise neighbor on the ring. A process receives, and sends along, a message with the temporary ID of its next clockwise neighbor. Therefore, a process also receives the temporary ID of its neighbor two steps clockwise. A process compares the three IDs.

- If the clockwise ID is largest, it replaces the temporary ID.
- If the clockwise ID is not the largest, the process goes into relay mode.
- If the clockwise ID is equal to the temporary ID, the (receiving) process is the leader.

The algorithm sends messages in only one direction (unidirectional) and does not require that the value of $n$ be known. For every process that stays active (because the middle ID was the largest), another process goes to relay mode (because its middle ID could not possibly be the largest). This cutting by two at each iteration yields logarithmic performance. The synchronous time complexity is $\log_2 n$ rounds and the message complexity is $O(n \log n)$.

Local Variables:

1. $temp_id \leftarrow i$, the temporary ID of the process
2. $next_temp_id$, the temporary ID of the next clockwise process
3. $next_next_temp_id$, the temporary ID of the next clockwise process
4. $neighbor \leftarrow i + 1$, the next counterclockwise process on the ring

Code: At $p_i \in \{1, ..., n\}$:

**ACTIVE:**

```
do forever
    send(neighbor, temp_id)
    next_temp_id \leftarrow \text{receive()}
    \text{if} \ next_temp_id = temp_id \text{then}
        \text{announce(LEADER)}
        send(neighbor, next_temp_id)
        next_next_temp_id \leftarrow \text{receive()}
    \text{if} \ next_temp_id > \text{max}(temp_id, next_next_temp_id)
        \text{then} \ temp_id \leftarrow next_temp_id
    \text{else} \ \text{goto RELAY}
```

**RELAY:**

```
do forever
    temp_id \leftarrow \text{receive()}
    send(neighbor, temp_id)
```
13A. Algorithm: Flood Leader Election

Outline: LeLann’s Leader Election algorithm (find the largest ID) can also be applied in a general network. In the Flood algorithm, the approach is the same (find the largest ID) but the structure is similar to Shortest Path. There are different versions depending on the information, or common knowledge, held by the nodes. If the nodes know the diameter (diam) of the network, then the largest ID can be discovered by all nodes in diam rounds. In the version below, the nodes only know n, the size of the network, and take n – 1 rounds. At each round, each node broadcasts its current estimate of the leader. It then receives the estimates from its neighbors and updates its own estimate if it sees a larger ID. The synchronous time complexity is n – 1 rounds and the message complexity is (n – 1)|E| = O(n^3).

Local Variables:

1. r, integer 1..n-1, the current round
2. leader, initially i, the estimate of the leader by node i

Code: At \( p_i \in \{1..n\} \):

```plaintext
for all rounds do {
    broadcast leader to all neighbors
    for all neighbors do {
        receive id from neighbor j
        if leader < id then
            update leader
    }
}
if leader = i then
    node i is the leader
else
    node i is not the leader
```
V. Common Knowledge

Given: \( n \) processes

**Problem:** The \( n \) processes must reach *common knowledge* concerning some true fact in the system. Common knowledge is “everyone knows that everyone knows,..., a fact”. This is difficult, and sometimes impossible, to achieve in a distributed system where messages are used to provide opinions as to the fact being true or false. Assume that process \( p_i \) learns of a fact and sends a message concerning this to process \( p_j \). Eventually, \( p_j \) gets the message and *knows* the fact but \( p_i \) does not know that \( p_j \) knows this yet. Therefore, \( p_j \) must send a message back to \( p_i \) (an acknowledgement) that it knows the fact now. Eventually, this message arrives at \( p_i \) and it knows that \( p_j \) knows the fact. The problem is that \( p_j \) does not know that \( p_i \) knows that \( p_j \) knows the fact. Process \( p_i \) can send another message stating its current knowledge but the problem is infinite because of the delay in the exchange of information. The only way to achieve common knowledge is for all parties involved (or processes) to achieve it at the same time.

**Example: Muddy Children Problem.** One rainy day, a bunch of children went outside to play and some of their faces became dirty. Eventually, their mother came out and saw that some faces were dirty and she said “Someone is dirty”. She asked them, “Do you know if your face is CLEAN or DIRTY?” Initially, all of the children are NOT_SURE because they cannot see their own face, but they can count how many other children are dirty. Assume there are \( k \) dirty faces. The mother keeps asking the same question for round after round. All children respond NOT_SURE for \( k - 1 \) rounds. In round \( k \), all of the dirty children respond, in unison, that their faces are DIRTY. The clean children, who just said they were NOT_SURE, immediately change their statement to CLEAN. The children have common knowledge of the true state of the world.

**Proof:** By induction on \( k \). If \( k = 1 \) then the single dirty child does not see any other dirty faces. When the mother says “Someone is dirty” then the dirty child knows they are the one.

Consider \( k > 1 \) and consider one of the dirty children, Nancy. Nancy can see \( k - 1 \) other children that are dirty, but cannot see whether or not her face is dirty. She knows that there can be either \( k - 1 \) total dirty faces (and that her own face is clean) or \( k \) total dirty faces (and that her own face is dirty). For round \( i < k \), Nancy will respond NOT_SURE, as will all the other dirty (and clean) children. She must repond NOT_SURE because she cannot prove that she is dirty. The other children would have given the same response in the rounds preceding \( k \) regardless of whether she was clean or dirty. But now consider round \( k \). If Nancy were clean, there would be only \( k - 1 \) dirty children, and they would have all answered DIRTY in round \( k - 1 \) (by our inductive hypothesis). Since this did not happen in round \( k - 1 \), Nancy now knows that it is not possible that she is clean. She knows she is dirty, and answers DIRTY in round \( k \), as do all the other \( k - 1 \) dirty children, who perform the same reasoning process as Nancy.

Remember that the clean children have an accurate count of the dirty children, although they do not know this in advance of round \( k \). When all of the \( k \) dirty children respond DIRTY in round \( k \), the clean children know they are safe, by the above inductive hypothesis. Therefore, they quickly change their minds to CLEAN and act as if they had done nothing wrong.

The children achieve common knowledge because they know exactly what happened the last round because there is no delay in the information. Also, the above is not really an algorithm. Instead, the children reach common knowledge because of their capacity to reason and because they assume that all of the children have the same capacity, so therefore, they all reach the correct conclusion at the same time.

In a distributed system, the concept of DIRTY translates to faulty or unreliable, say a bad CPU, memory, or communication channel.
VI. Distributed Snapshots

**Given:** Processes on a network, each with their own local state variables, and messages on channels containing information.

**Problem:** What is the current global state, that is, the composite of the local states and messages? This information should be available to all processes. The state should be consistent in that some order of logical time could yield this state. It does not mean that this corresponds to real time. The snapshot state should reachable from the initial state and the final state should be reachable from the snapshot.

Consider a single-token conservation system and two processes, \( p \) and \( q \), connected by channels \( C \) and \( D \), as shown in Figure 3. If a process has the token, it is in local state \( S_1 \), otherwise state \( S_0 \). A process in \( S_1 \) can send the token on a channel, and it changes state to \( S_0 \). When a process, in \( S_0 \), receives the token, it changes state to \( S_1 \). The state of system can be described on the basis of the location of the token: \( \text{in}_p, \text{in}_C, \text{in}_q, \text{in}_D \).

![Diagram of Single-Token Conservation System]

Figure 3: Single-Token Conservation System

There is a problem with when and how to record the global state with respect to the recording of the local process and channel states. Assume that process \( p \) has the token and records this fact such that the global state is marked as \( \text{in}_p \). Then \( p \) sends the token on channel \( C \) and, at some point, the state of the channel is recorded such that the global state is marked as \( \text{in}_C \). There appear to be two tokens in the system. The problem is that the state of \( p \) is recorded before \( p \) sent the message along \( C \) and the state of \( C \) is recorded after \( p \) sent the message.

Define:

1. \( n = \) the number of messages sent along \( C \) (\( D \)) before recording the state of \( p \) (\( q \))
2. \( n' = \) the number of messages sent along \( C \) (\( D \)) before recording the state of \( C \) (\( D \))
3. \( m = \) the number of messages received along \( C \) (\( D \)) before recording the state of \( q \) (\( p \))
4. \( m' = \) the number of messages received along \( C \) (\( D \)) before recording the state of \( C \) (\( D \))

The problem with the above scenario is that \( n < n' \) (\( 0 < 1 \)).
Now consider the alternate scenario. Suppose the state of \( C \) is recorded as not possessing the token and then \( p \) sends the token on \( C \) and records its state as not possessing the token. The system appears to have no tokens, the problem is that \( n > n' \). Therefore, we deduce that a consistent global state requires \( n = n' \), that is, the states of \( p \) and \( C \) must be recorded after an equal number of messages.

A similar argument can be made that \( m = m' \). In every state, the number of messages received along a channel cannot exceed the number of messages sent along that channel, that is, \( n' \geq m' \). From these equations, \( n \geq m \). The state of channel \( C \) that is recorded must be the sequence of messages sent along the channel before the sender’s state is recorded, excluding the sequence of messages received along the channel before the receiver’s state is recorded.

This can be accomplished by \( p \) sending a special \textit{marker} after it records its state and before sending any other messages (e.g., the token). The process at the receiving end of a channel is responsible for recording the state of the incoming channel. In the above example, \( q \) records the state of \( C \) as the sequence of messages received by \( q \) after \( q \) records its own state and before \( q \) receives the marker along \( C \).

Formally, the rules are:

**RULE 1:** Any process recording its own state will send a special marker (\#) along an outgoing channel \( C \):

- after recording its own state
- before sending anything else along channel \( C \)

**RULE 2:** The state of a channel \( C \) is recorded by process \( q \) as the sequence of messages received by \( q \) along \( C \):

- after recording \( q \)'s own state
- before receiving the marker along channel \( C \)

**RULE 3:** Upon receiving a marker along an incoming channel \( C \), a process \( q \) should do the following:

- case1 (\( q \) has already recorded its own state): Then \( q \) records the state of \( C \) as the sequence of messages received after it recorded its state and before it received the marker.

- case2 (\( q \) has not already recorded its own state): Then \( q \) records its own state, and records the state of \( C \) as empty.

Instead of a single token, consider a system which has some fixed number of tokens or DOLLARS (\$). Let both \( p \) and \( q \) start off with \$1 and call this Global START State \( A_0 \). A consistent global snapshot should always show the system with a total of \$2. These dollars can be transmitted in messages along channels as before. The algorithm proceeds as follows (see Figure 4):

1. \( p \) spontaneously records its own state as \$1
2. \( p \) sends a marker to \( q \) along \( C \) because it just recorded its own state
3. \( p \) sends \$1 to \( q \)

The Global State is now \( A_1 \): the marker and the \$ are in channel \( C \).

4. \( q \) sends \$1 to \( p \) along channel \( D \)
5. $q$ receives the marker from $p$ along $C$
6. $q$ records the state of $C$ as empty
7. $q$ records its own state as $0$ (because it has not recorded it yet)
8. $q$ sends a marker to $p$ because it just recorded its own state

The Global State is now $A_2$: the $@$ is still in $C$ and there is a $@$ and marker in $D$.

9. $p$ receives $1$ from $q$ along $D$
10. $p$ receives the marker from $q$ along $D$
11. $p$ records the state of $D$ as $1$

The Global State is now FINAL $A_3$: $p$ has $1$ and channel $C$ has $1$.
The Global SNAPSHOT is the state: $p$ has $1$ and $D$ has $1$.

Note that the SNAPSHOT is not the same as the FINAL state. However, it is consistent in that a total of $2$ is in the system and that the SNAPSHOT can be reached from the START state and the FINAL state is reachable from the SNAPSHOT.

The FINAL state reflects the events that took place:

1. $p$ sent and received $1$
2. $q$ sent $1$ but has not received any money

The SNAPSHOT state reflects the states that were recorded:

3. $p$ has $1$
4. $q$ has $0$
5. $D$ has $1$

The SNAPSHOT state is reachable from the START state assuming:

6. $q$ has sent $1$ along $D$ but it has not been received

The FINAL state is reachable from the SNAPSHOT state assuming:

7. $p$ has sent $1$ and has received $1$

The assumptions 6 and 7 are exactly what took place in the system but with a different ordering of the logical events.
Figure 4: Global Snapshot in a Two-Dollar Conservation System
Figure 5: Time-Space Diagram of Application and Snapshot Events

Figure 6: Warped Time-Space Diagram of Application Events
SUMMARY OF ALGORITHM:

DEFINE:
current state of node p: p
current state of channel C: C
snapped state of node p: p'
snapped state of node channel C: C'

GOAL: snap own state p', snap all incoming channels C_j
INITIATOR: snap own state, send # on all outgoing channels

ALG:
Upon 1st # on C_j: snap own state, snap C_j = empty, send # on all outgoing channels
Upon Application Msg on C_j: act on message but save in special queue Q_j
Upon other # on C_j: snap C_j = Q_j

<table>
<thead>
<tr>
<th>n (sent before snap p')</th>
<th>n' (sent before snap C')</th>
</tr>
</thead>
<tbody>
<tr>
<td>p' = $ send $ C' = $</td>
<td>0           1</td>
</tr>
<tr>
<td>C' = empty send $ p' = 0</td>
<td>1           0</td>
</tr>
<tr>
<td>p' = $ C' = empty send $</td>
<td>0           0</td>
</tr>
<tr>
<td>send $ p' = 0 C' = $</td>
<td>1           1</td>
</tr>
</tbody>
</table>

Figure 7: Proof that n = n'

Summary: n ≥ m ⇒ n' = n ≥ m = m'
VII. Consensus/Byzantine Generals

Given: $n$ processes on a network with $t$ of them faulty. That is, the processes have halted or are just not communicating.

Problem: The non-faulty processes must reach a consensus, or agreement, among themselves about the value of a bit: 0 or 1. The agreement, which may take several phases of communication, allows the non-faulty processes to continue to cooperate. Assume that every process starts with an initial value from \{0, 1\}. A process decides on a value in \{0, 1\} by entering an appropriate decision state. The requirements for a solution are as follows:

1. Agreement: No two non-faulty processes may decide on different values.

2. Validity: If all non-faulty processes have the same initial value, then no other value may be decided upon by a non-faulty process.

3. Termination: All non-faulty processes must eventually decide.

In a synchronous network with at most $t$ faulty processes and no authentication of messages, consensus can be reached only if $n \geq 3t + 1$. With authentication of messages, any number of traitors can be handled. In any case, it will take at least $t + 1$ rounds in a synchronous protocol to guarantee a solution.

One variation, the Byzantine Generals, is a leader (or Commanding General) which broadcasts a bit first (or some arbitrary value) and the other Generals try to reach agreement. Note that the Commanding General might also be faulty.
Figure 8: A traitor lieutenant can deceive two loyal generals.

Figure 9: A traitor commander can deceive two loyal generals.
Figure 10: A traitor lieutenant cannot deceive three loyal generals.

Figure 11: A traitor commander cannot deceive three loyal generals.
VIII. Distributed Minimum-Weight Spanning Tree

**Given:** \( n \) processes on network nodes with a distinct cost for edge \((i,j)\), perhaps infinite, between node \( i \) and \( j \).

**Problem:** The processes must cooperatively find the Minimum-Weight Spanning Tree (MST). That is, the tree that connects all processes (nodes) at the minimum cost.

(a) Original network

(b) Initial fragments with level \( LN=0 \)

(c) Merge fragments (\( FN=\text{core} \)) with equal \( LN \); \( LN++ \)

(d) Merge again

(e) MST: large fragment (\( LN=2 \)) absorbs small fragment (\( LN=1 \)) and keeps \( FN, LN \)
Algorithm: Gallager-Humblett-Spira's MST

Outline: (From: MIT Lecture Notes) The central idea is that nodes form themselves into collections -fragments- of increasing size (see the figure on the previous page). Initially, all nodes are considered to be in singleton fragments. Each fragment is itself connected by edges that form a MST for the nodes in the fragment. Within any fragment, nodes cooperate in a distributed algorithm to find the Minimum-Weight Outgoing Edge (MWOE) for the entire fragment (that is, the minimum weight edge that leads to a node outside the fragment). It is well-known that two fragments, each with an MST inside, can be joined by the MWOE to form a MST for the merged fragment. This property is the basis for all sequential and distributed MST algorithms. The strategy for accomplishing this involves broadcasting over the edges of the fragment, asking each node separately for its own MWOE leading outside the fragment. Once all these edges have been found, the minimal edge among them will be selected as an edge to include in the (eventual) MST. Once a MWOE for a fragment is found, a message may be sent out over that edge to the fragment on the other side. The two fragments may then combine into a new, larger fragment. The new fragment then finds its own MWOE, and the entire process is repeated until all the nodes in the graph have combined themselves into one giant fragment (whose edges are the MST).

This is not the whole story, of course; there are some problems to overcome. First, how does a node know which of its edges lead outside its current fragment? A node in fragment \( FN \) can communicate over an outgoing edge, but the node at the other end needs some way of telling whether it too is in \( FN \). We will therefore need some way of naming fragments (the solution: use the weight of the core edge (MWOE) that was used to combine two smaller fragments). But the issue is still more complicated: it may be, for example, that the other node (at the end of the apparently outgoing edge) is in \( FN \) but hasn’t learned this fact yet because of communication delays. Thus, some sort of overall synchronization process is needed. This will be a two-phase strategy that ensures that nodes won’t search for outgoing edges until all nodes in the fragment have been informed of their current fragment.

Another problem is that the number of messages sent by such an algorithm could be large. The number of messages sent by a fragment to find its MWOE will be proportional to the number of nodes in the fragment. Under certain circumstances, one might imagine the algorithm proceeding by having one large fragment that picks up a single node at a time. In such a situation, the algorithm would require \( O(n^2) \) messages to be sent.

This second problem should suggest a “balanced-tree algorithm” solution: that is, the difficulty derives from the merging of data structures that are very unequal in size. The strategy that we will use, therefore, is to merge fragments of roughly equal size. Intuitively, if we can keep merging fragments at nearly equal size, we can keep the number of total messages to \( O(n \log n) \).

The trick we will use to keep the fragments of similar sizes is to associate level numbers \( LN \) with each fragment. If the level is \( LN \) for a given fragment, then the number of nodes in the fragment \( \geq 2^{LN} \). Initially, all fragments are just singleton nodes at \( LN = 0 \). When two fragments at level \( LN \) are merged together, the result is a new fragment with \( LN = 1 \). (This preserves the condition that we specified for level numbers: if two fragments of size at least \( 2^{LN} \) are merged, the result is a new fragment of size at least \( 2^{LN+1} \).)

There are two ways of combining fragments:

1. **Merging.** This is the “standard” way of combining. The rules are:
   a. Both fragments must have the same value for \( LN \).
   b. Both fragments must share the same MWOE.
   c. The fragments are combined using the MWOE (core edge).
   d. The new \( LN \) is just the old \( LN + 1 \).
e. The new $FN$ is the (distinct) weight of the MWOE.
f. If a large fragment finds that its MWOE leads to a smaller fragment (small $LN$), then the larger fragment holds up and takes no action.

2. Absorbing. It might be that some nodes are forming into huge fragments via merging, but isolated (or small fragments) are lagging behind at a low level ($LN$). In this case, a small fragment may be absorbed into the larger one without determining the MWOE of the large fragment. The rules are:

a. Let $FN$ be the larger fragment (large $LN$)
b. The other fragment’s MWOE must lead to $FN$
c. The fragments are combined using the MWOE of the smaller fragment.
d. The combined fragment keeps the larger fragment’s $SN$ and $FN$ (same core edge).
e. The combined fragment is not “new”, just augmented.

The following message types are used in the code:

- **Initiate** messages are broadcast outward on the edges of a fragment to tell nodes to start finding their MWOE.
- **Report** messages are the messages that send the MWOE information back in (these represent the convergecast response to the Initiate broadcast messages).
- **Test** messages are sent out by nodes when they search for their own MWOE.
- **Accept** and **Reject** messages are sent in response to Test messages from nodes; they inform the testing node whether the responding node is in a different fragment (Accept) or is in the same fragment (Reject).
- **Change_root** is a message sent toward a fragment’s MWOE once that edge is found. The purpose of this message is to change the root of the (merging or absorbing) fragment to the appropriate new root.
- **Connect** messages are sent across an edge when a fragment combines with another. In the case of a merge operation, Connect messages are sent both ways along the edge between the merging fragments; in the case of an absorb operation, a Connect message is sent by the “smaller” fragment along its MWOE toward the “larger” fragment.
- **Wakeup** messages can be used to spontaneously start a node.
- **Halt_all** messages are used when a node discovers that the MST is finished.

In a bit more detail, Initiate messages emanate outward from the designated “core edge” to all nodes of the fragment; these Initiate messages not only signal the nodes to look for their own MWOE (if that edge has not yet been found), but they also carry information about the fragment identity (the core edge) and the level number of the fragment. As for the Test-Accept-Reject protocol: every node, in order to avoid sending out redundant messages testing and retesting edges, sends messages along edges in the order of the weights. The nodes classify these incident edges in one of three categories:

- **Branch** edges are those edges designated as part of the building MST.
- *Basic* edges are those edges that the node doesn’t know anything about yet - they may yet end up in the MST. (Initially, all the node’s edges are classified as *Basic*.)

- *Rejected* edges are edges that cannot be in the MST (i.e., they lead to another node within the same fragment).

A fragment node searching for its MWOE need only send messages along *Basic* edges. The node tries each edge in order, lowest weight to highest. The protocol is to send a *Test* message with the *FN* (unique weight of the *core* edge) and *LN*. The recipient of the *Test* message then checks if its own identity is the same as the tester; if so, it sends back a *Reject* message. If the recipient’s identity (core edge) is different and its level is greater than or equal to that of the tester, it sends back an *Accept* message. Finally, if the recipient has a different identity from the tester but has a lower level number, it delays responding until such time as it can send back a definite *Reject* or *Accept*. The delay is achieved by a *requeue* of the message to the end of the queue.

When two *Connect* messages cross, this is the signal that a merge operation is taking place. In this event, a new *Initiate* broadcast emanates from the new core edge and the newly-formed fragment begins once more to look for its overall MWOE. If an absorbing *Connect* occurs, from a lower-level to a higher-level fragment, then the node in the high-level fragment know whether it has found its own MWOE and thus whether to send back a *Initiate* message to broadcast in the lower-level fragment.
Local Variables:

1. $SN \in \{Sleeping, \ Find, \ Found\}$, initially $Sleeping$, is the state of the node.

2. $SE$: integer array $1..n$ from $\{Basic, \ Branch, \ Rejected\}$, initially $Basic$, where $SE[j]$ is the state of the edge to node $j$.

3. $FN$: integer from the set of edge weights is the fragment identity.

4. $LN$: integer from $\{0..\log_2 n\}$ is the level.

5. $best\_edge, \ test\_edge, \ in\_branch$: integers from the set of edges.

6. $best\_wt$: integer from the set of weights

7. $find\_count$: integer.

8. $w$: integer array $[1..n]$ where $w[j]$ is distinct cost, perhaps infinite, of the edge from $i$ to $j$.

**Code:** At $p_i \in \{1,...,n\}$:

1. Response to receipt of $Wakeup()$
   
   ```
   execute procedure wake_up
   ```
   
2. **procedure** $wake\_up$
   
   ```
   m \leftarrow \min_j w(j)
   SE[j] \leftarrow Branch; \ LN \leftarrow 0; \ SN \leftarrow Found; \ find\_count \leftarrow 0
   send Connect(0) on edge $j$
   ```

3. Response to receipt of $Connect(L)$ on edge $j$
   
   ```
   if $SN = Sleeping$ then execute procedure $wake\_up$
   ```
   
   ```
   if $L < LN$ then { /* absorb */
   SE[j] \leftarrow Branch
   send $Initiate(LN, FN, SN)$ on edge $j$
   if $SN = Find$ then $find\_count \leftarrow find\_count + 1$
   }
   ```

   ```
   else if $SE[j] = Basic$ then requeue $Connect(L)$ on edge $j$
   else send $Initiate(LN + 1, w[j], Find)$ on edge $j$ /* merge */
   ```

4. Response to receipt of $Initiate(L, F, S)$ on edge $j$
   
   ```
   LN \leftarrow L; \ FN \leftarrow F; \ SN \leftarrow S
   in\_branch \leftarrow j \ best\_edge \leftarrow nil; \ best\_wt \leftarrow \infty
   ```
   
   ```
   for $k = 1,...,j - 1,j + 1,...,n$ do
   if $SE[k] = Branch$ then {
   send $Initiate(L, F, S)$ on edge $k$
   if $S = Find$ then $find\_count \leftarrow find\_count + 1$
   }
   ```

   ```
   if $S = Find$ then execute procedure $test$
   ```
5. procedure test
   \[ m \leftarrow \min_j w(j) \text{ where } SE[j] = \text{Basic} \]
   if \( m \neq \infty \) then {
      test_edge \leftarrow j
      send \ Test(LN, FN) \text{ on edge test_edge} \}
   else {
      test_edge \leftarrow \text{nil}
      execute procedure report
   }

6. Response to receipt of Test(L, F) on edge j
   if \( SN = \text{Sleeping} \) then execute procedure wake up
   if \( L > LN \) then requeue Test(L, F) on edge j
   else if \( F \neq FN \) then send Accept() on edge j
   else {
      if \( SE[j] = \text{Basic} \) then \( SE[j] \leftarrow \text{Rejected} \)
      if test_edge \neq j then send Reject() on edge j
      else execute procedure test
   }

7. Response to receipt of Accept() on edge j
   test_edge \leftarrow \text{nil}
   if \( w[j] < \text{best wt} \) then {
      best_edge \leftarrow j; best_wt \leftarrow w[j]
   }
   execute procedure report

8. Response to receipt of Reject() on edge j
   if \( SE[j] = \text{Basic} \) then \( SE[j] \leftarrow \text{Rejected} \)
   execute procedure test

9. procedure report
   if \( \text{find count} = 0 \) and test_edge = \text{nil} \ then {
      \( SN \leftarrow \text{Found} \)
      send Report(best_wt) on edge in_branch
   }
10. Response to receipt of \texttt{Report}(w) on edge \texttt{j}
\[
\text{if } j \neq \text{in\_branch} \text{ then } \{ \\
\quad \text{find\_count} \leftarrow \text{find\_count} - 1 \\
\quad \text{if } w < \text{best\_wt} \text{ then } \{ \\
\quad\quad \text{best\_wt} \leftarrow w; \text{best\_edge} \leftarrow j \\
\quad \}\text{execute procedure report} \\
\}\text{else if } SN = \text{Find then requeue Report}(w) \text{ on edge } j \\
\text{else if } w > \text{best\_wt} \text{ then execute procedure change\_root} \\
\text{else if } w = \text{best\_wt} = \infty \text{ then } \{ \\
\quad \text{send Halt\_all}() \text{ on all edges } j; \text{halt} \\
\}\]

11. \textbf{procedure change\_root} \\
\text{if } SE[\text{best\_edge}] = \text{Branch then send Change\_root}() \text{ on edge } best\_edge \\
\text{else } \{ \\
\quad \text{send Connect}(LN) \text{ on edge } best\_edge \\
\quad SE[best\_edge] \leftarrow \text{Branch} \\
\}\]

12. Response to receipt of \texttt{Change\_root}() \\
\textbf{execute procedure change\_root} \\

13. Response to receipt of \texttt{Halt\_all}() \\
\text{send Halt\_all}() \text{ on all edges } j; \text{halt} \\

14. Main body \\
\textbf{do forever} \\
\text{receive msg on edge } j \\
\textbf{case} msg.type is \\
\quad \text{Wakeup then respond to Wakeup}() \\
\quad \text{Connect then respond to Connect}(msg.L) \text{ on edge } j \\
\quad \text{Initiate then respond to Initiate}(msg.L, msg.F, msg.S) \text{ on edge } j \\
\quad \ldots \\
\textbf{endcase}
Algorithm: Breadth-First Search Spanning Tree

Outline: A root node broadcasts a search message to all neighbors. If a node has not been marked, it accepts the sender as the parent and broadcasts to all neighbors. Termination is detected using a "convergecast". Each node acknowledges a search message with either a parent or non-parent message and, therefore, a node knows all of its children. If a node has received acknowledgements for all search messages and completion messages from all of its children, then the node sends a completion message to its parent. The synchronous time complexity is $O(diam) = O(n)$ and the message complexity is $O(|E|) = O(n^2)$.

Local Variables:

1. marked, boolean, initially false
2. parentNode, initially -1, identity of current node’s parent
3. searchCount, initially 0, number of search messages sent out
4. searchResponses, initially 0, number of responses to search messages
5. childCount, initially 0, number of children of current node
6. completeReceived, initially 0, number of completed messages received
7. $w$: integer array $[1..n]$ where $w[j]$ is distinct cost, perhaps infinite, of the edge from $i$ to $j$.

Code: At $p_i \in \{1..n\}$:

root broadcasts SEARCH message to all neighbors

**do forever**

receive $msgType$ from process $j$

**case** $msgType$ is

PARENT, NONPARENT, COMPLETE **then** maintain counters

SEARCH **then**

if marked **then** send NONPARENT to $j$

else {

marked ← true

parentNode ← $j$;

send PARENT to $j$

broadcast SEARCH to all neighbors except $parentNode$

}

endcase

if all search messages acknowledged **then**

if all children are completed **then**

send COMPETE to $parentNode$ and STOP
**Algorithm: Bellman-Ford Shortest Path**

**Outline:** MST minimizes the sum of the tree edges and SP minimizes the length of the path from the root to each node. If the weight on an edge represents delay, then the maximum of the shortest paths determines the time for the root to communicate to all nodes. A SP tree and MST tree may, or may not, be the same for a given graph. In the example given in MST, the graph does yield the same MST and SP tree. Note that a BFS tree is a specific example of both MST and SP when the edges have a cost of 1.

Bellman-Ford SP is either synchronous (processing by rounds) or asynchronous. The synchronous version given here knows that it is done after \( n - 1 \) rounds. At each round, each node \( i \) broadcasts to its neighbors its own current estimate of the shortest path from the root to \( i \). Each node receives all of the estimates and checks if these can help to produce even a shorter path. After a particular round \( r \), each node knows the shortest path from the root involving \( r \) edges or less. The synchronous time complexity is \( n - 1 \) rounds and the message complexity is \( (n - 1)|E| = O(n^2) \).

**Local Variables:**

1. \( r \), integer 1..n-1, the current round
2. dist, initially 0 at root (\( i = 1 \)) and \( \infty \) elsewhere, the current shortest path to the root
3. parentNode, initially -1, identity of node toward current shortest path to root
4. \( w \): integer array [1..n] where \( w[j] \) is distinct cost, perhaps infinite, of the edge from \( i \) to \( j \).

**Code:** At \( p_i \in \{1..n\} \):

```
for all rounds do {
    broadcast dist to all neighbors
    for all neighbors do {
        receive d from neighbor j
        if \( d + w[j] < \) dist then
            update dist and parentNode
    }
}
```
Algorithm: Asynchronous Shortest Path

Outline: One problem with the Synchronous Shortest Path is that the number of nodes $n$ must be known in advance. Instead, if a message is received, then check if this improves the current estimate of the shortest path. If so, then do the usual update but also broadcast this new estimate to all neighbors. Note that the algorithm does not have $n-1$ synchronous rounds or a receive from all neighbors. To get the algorithm started, the root broadcasts its shortest path (0) to all of its neighbors.

The problem with the asynchronous version is that the nodes do not know when the algorithm is terminated. They each have the best shortest path but they do not realize this fact. A termination detection algorithm (see Section X) can be added to the above algorithm.
IX. Logical Clocks

**Given:** \( n \) processes which do not share a physical clock and events on a network (broadcast, send, receive, local event).

**Problem:** What is the ordering of events? That is, did event \( a \) occur before event \( b \) or vice versa.

Logical clocks can be used in place of physical clocks (which will never be totally synchronized anyway). A Partial Order Logical Clock has a “happened before” relation (\( \rightarrow \)). If \( a \) occurred before \( b \), then \( a \rightarrow b \). But if \( c \rightarrow b \), this does not tell us if \( a \rightarrow c \) or \( c \rightarrow a \). There are two type of events that can help with partial order.

1. if \( a \) and \( b \) are events in the same process, and \( a \) comes before \( b \), then \( a \rightarrow b \).

2. if \( a \) is the sending of a message by one process and \( b \) is the receipt of the same message by another process, then \( a \rightarrow b \).

For example, Figure 12 shows that process 2 receives a message from both process 1 and process 3. Obviously the sending events must have occurred before the receiving events. But it is unknown which of the sending events occurred first and these events are considered “concurrent”.

![Figure 12: The Partial Ordering of Events based on Message Sending and Receiving. Total Ordering: \( b \Rightarrow a \Rightarrow c \Rightarrow d \Rightarrow e \Rightarrow f \)]

A counter is used at each process to implement a logical clock. Let \( c_i(a) \) be the time event \( a \) occurred at process \( i \) and let the function \( C \) represent the entire system of clocks. Therefore, \( C(a) = c_i(a) \) if event \( a \) took place at process \( i \). It is important that the logical clocks guarantee the Clock Condition: if \( a \rightarrow b \) then \( C(a) < C(b) \).
Algorithm: Lamport’s Partial Order Logical Clock

Outline: The counter $c_i$, initially 0, is used as a timestamp and should be included in every message.

Lamport Time Transformation:
before send event or local event: $c_i++$
after receive event: $c_i++$ and ensure $c_i >$ timestamp in message, say by 1
after all adjustments for event $a$: $c_i(a) = c_i$

Local Variables:

1. counter : the timestamp

Code: At $p_i \in \{1, 2\}$:

\[
\text{do forever} \\
\text{process some local events} \\
\text{send a message with a timestamp} \\
\text{receive a message with a timestamp} \\
\text{maintain the counter}
\]

SUMMARY OF TOTAL ORDERING ALGORITHM

Define:
$c_i$: node i’s counter.
$c'_i$: node i’s counter at the moment of “trying” to get critical section.
$c_j$: node i’s “view” of node j’s logical time.

$c_i = 0, c'_j = \infty, c'_j = -\infty \\
c'_i = c_i, \ TRY(c'_i) \\
\forall j \neq i \ c'_j > c'_i \wedge c'_j > c'_i$

\[
\begin{array}{c}
\text{INIT} \rightarrow \text{NOT TRYING} \rightarrow \text{TRYING} \rightarrow \text{GRANTED} \\
\end{array}
\]

$\text{EXIT}(c_i)$

$\text{TRY}(c'_i)$

\[
\begin{array}{c}
i \\
\text{ACK}(c_j), c'_j = c_j \\
\end{array}
\]
Algorithm: Lamport’s Total Order Logical Clock for Mutual Exclusion

Outline: The partial order of logical clocks can be extended to a total order by using the process ID to break ties: If $a$ is an event in process $p_i$ and $b$ is an event in process $p_j$, then $a \Rightarrow b$ if and only if either (i) $c_i(a) < c_j(b)$ or (ii) $c_i(a) = c_j(b)$ and $p_i < p_j$.

The total order can be used to achieve mutual exclusion on a network. Each process broadcasts that it is “trying” to get into its critical section and each process “acknowledges” the receipt of this message. Only one process is allowed into the critical section based on timestamps included in the messages. When the one process is finished with the critical section, it broadcasts an “exit” message. Note that a broadcast is an “event” itself and not each individual message.

If the loop below is finite, then the first process which stops also needs to broadcast a “stop” message to all other processes. This allows these processes to stop and not to expect acknowledgements. Lamport’s algorithm maintains queues of received messages to replicate a global centralized queue that determines the service order. However, the algorithm below does not use queues but just the most recent message information.

Local Variables:

1. counter $c_i$: current timestamp
2. my.trying.time $c^i_i$: timestamp
3. $j$’s trying.time $c^j_j$: array of timestamp, initially infinity
4. view of $j$’s local.time $c^j_j$: array of timestamp, initially negative infinity
5. trying, granted: boolean, initially false

Code: At $p_i \in \{1, ..., n\}$:

\[
\text{do forever}
\]

\[
\text{if } \neg \text{trying} \text{ then } \{
\text{trying } \leftarrow \text{true}
\text{record my.trying.time}
\text{broadcast a TRY message}
\}
\]

\[
\text{if trying then } \{
\text{granted } \equiv \forall (j \neq i) \text{ j’s trying.time } > \text{my.trying.time and}
\text{view of j’s local.time } > \text{my.trying.time}
\text{if granted then trying } \leftarrow \text{false}
\}
\]

\[
\text{if granted then } \{
\text{enter and exit critical section}
\text{broadcast an EXIT message}
\text{granted } \leftarrow \text{false}
\}
\]
receive message from $j$ and maintain counter
maintain view of $j$’s local_time

**case** message type is

TRY **then** maintain $j$’s trying_time and acknowledge
EXIT **then** reset $j$’s trying_time to infinity
ACK **then** do nothing

**endcase**
X. Termination Detection and Stable States

**Given:** Asynchronous application algorithm.

**Problem:** When is the algorithm terminated?

For example, a synchronous shortest-path algorithm knows it is done after \( n - 1 \) rounds. An asynchronous shortest-path algorithm checks its best path every time it receives a message but a node does not know when it will never receive any more messages again. The algorithm is essential over when all of the nodes are quiescent and no more messages are travelling on the network. Each node knows its current shortest-path to the root but does not know that this is really the ultimate value. The algorithm remains in this terminated state forever, and this is called a **stable state**. Likewise, a deadlocked algorithm is also in a stable state.

There are two main ways to detect termination. The general solution is to run, periodically, a Global Snapshot as a sub-process. The application may be in a stable state but the current snapshot may not show this fact due to the limitations of snapshots. However, a subsequent snapshot will eventually show the stable state (and so will any more snapshots).

To detect a termination stable state, another solution is to run a specific termination detection algorithm, such as Dijkstra-Scholten. The idea is to maintain a spanning tree which grows and shrinks with the message activity of the underlying application algorithm. The detection sub-process only requires the addition of ACK messages and does not interfere with the underlying application.

If a quiescent node receives an application message, then it remembers the sender as its parent - just like a BFS. If the node is already awake, then it sends an ACK. Whenever an application sends out messages, it counts how many are outstanding (i.e., no ACK yet). Whenever a node goes quiescent, it checks to see if it has received all of its ACKs. If so, then it can ACK its parent and shrink the tree by setting its parent value to null. Likewise, the parent might shrink the tree. The ACKs are a convergecast to the application messages. The tree might not shrink all the way to the root yet because non-quiescent nodes might start a new flurry of message broadcasts. Previously quiescent nodes will receive messages and expand the tree.

One important state variable at each node is the boolean **quiescent**. A spanning tree can only shrink if the node is quiescent and it has received ACKs for all application messages. For example, an asynchronous shortest-path algorithm is quiescent before its receives an application message, then non-quiescent as it sends updates to its neighbors, and then quiescent again.

Termination is detected at the root when it receives all ACKs and it is quiescent itself. Note that this detection algorithm assumes that the underlying application is “source” based. That is, only one node wakes up, say to read input, and then starts running.

See Figure 13 and Figure 14.

One example is the Asynchronous Shortest Path algorithm (21B). The termination detection algorithm with can be integrated with the normal application messages so that the root knows when the algorithm is done (and can broadcast this fact).
Figure 13: Outline of Dijkstra-Scholten Algorithm
Figure 14: Example of Dijkstra-Scholten Algorithm

M = Application Message  A = Acknowledgement
XI. Phased Algorithms
Distributed algorithms may be combined, either by sequential text or by integrating into one SWITCH statement.

24. Algorithm: BFS and Leader Election

PHASE 1: Starting from the root, construct a Breadth-First Search tree on a graph.
   a. After completion, each node “knows” its PARENT.
   b. One new feature: each node should remember its CHILDREN.

PHASE 2: After the BFS completes, the ROOT starts a leader election algorithm.
   a. The election algorithm should only use the BFS tree edges (parent and children).
   b. The nodes should elect the LARGEST ID as the leader.
   c. Use one SWITCH statement (better) or construct program in sequential phases.
   d. Two new message types might be: FINDMAX and REPORTMAX.
   e. At the end of this phase, the ROOT “knows” the MAX leader.
   f. Other nodes “know” the MAX in their sub-tree.
   g. All nodes know the PATH towards their sub-tree MAX (which child “points” to MAX).

PHASE 3: The root NOTIFIES just the nodes along the PATH towards the new leader.
   a. A node converts its PARENT to a CHILD.
   b. A node converts its PATH to a PARENT.
   c. The NOTIFY stops at the new leader in preparation for PHASE 4.
   d. The net effect is to redirect the tree edges to be ROOTED at the new LEADER.

PHASE 4: When the new leader ROOT gets the NOTIFY message,
   a. The leader sends out a NEWROOT message along the tree edges.
   b. This just tests the new redirected edges on the tree.