A set of processes coordinate their actions. How to agree on one or more values
- Avoid fixed master-salve relationship to avoid single points of failure for fixed master.

Distributed mutual exclusion for resource sharing
- A collection of processes share resources, mutual exclusion is needed to prevent interference and ensure consistency. (critical section)
- No shared variables or facilities are provided by single local kernel to solve it. Require a solution that is based solely on message passing.

Important factor to consider while designing algorithm is the failure
Distributed Mutual Exclusion

- Application level protocol for executing a critical section
  - `enter()` // enter critical section-block if necessary
  - `resourceAccess()` // access shared resources
  - `exit()` // leave critical section-other processes may enter

Essential requirements:
- **ME1**: (safety) at most one process may execute in the critical section
- **ME2**: (liveness) Request to enter and exit the critical section eventually succeed.
- **ME3**: (ordering) One request to enter the CS happened-before another, then entry to the CS is granted in that order.

ME2 implies freedom from both deadlock and starvation. Starvation involves fairness condition. The order in which processes enter the critical section. It is not possible to use the request time to order them due to lack of global clock. So usually, we use happen-before ordering to order message requests.
Performance Evaluation

- **Bandwidth** consumption, which is proportional to the number of messages sent in each entry and exit operations.

- The **client delay** incurred by a process at each entry and exit operation.

- **Throughput** of the system. Rate at which the collection of processes as a whole can access the critical section. Measure the effect using the **synchronization delay** between one process exiting the critical section and the next process entering it; the shorter the delay is, the greater the throughput is.
Central Server Algorithm

- The simplest way to grant permission to enter the critical section is to employ a server.
- A process sends a request message to server and awaits a reply from it.
- If a reply constitutes a token signifying the permission to enter the critical section.
- If no other process has the token at the time of the request, then the server replied immediately with the token.
- If token is currently held by other processes, the server does not reply but queues the request.
- Client on exiting the critical section, a message is sent to server, giving it back the token.
Figure 12.2
Central Server algorithm: managing a mutual exclusion token for a set of processes

ME1: safety
ME2: liveness
Are satisfied but not
ME3: ordering

Bandwidth: entering takes two messages (request followed by a grant), delayed by the round-trip time; exiting takes one release message, and does not delay the exiting process.

Throughput is measured by synchronization delay, the round-trip of a release message and
Ring-based Algorithm

- Simplest way to arrange mutual exclusion between N processes without requiring an additional process is arrange them in a logical ring.
- Each process pi has a communication channel to the next process in the ring, p(i+1)/mod N.
- The unique token is in the form of a message passed from process to process in a single direction clockwise.
- If a process does not require to enter the CS when it receives the token, then it immediately forwards the token to its neighbor.
- A process requires the token waits until it receives it, but retains it.
- To exit the critical section, the process sends the token on to its neighbor.
Figure 12.3
A ring of processes transferring a mutual exclusion token

ME1: safety
ME2: liveness
Are satisfied but not
ME3: ordering

Bandwidth: continuously consumes the bandwidth except when a process is inside the CS. Exit only requires one message

Delay: experienced by process is zero message (just received token) to N messages (just pass the token).

Throughput: synchronization delay between one exit and next entry is anywhere from 1 to N message transmission.
Using Multicast and logical clocks

- Mutual exclusion between N peer processes based upon multicast.
- Processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message.
- The condition under which a process replies to a request are designed to ensure ME1 ME2 and ME3 are met.
- Each process \( p_i \) keeps a Lamport clock. Message requesting entry are of the form \( <T, p_i> \).
- Each process records its state of either RELEASE, WANTED or HELD in a variable state.
  - If a process requests entry and all other processes is RELEASED, then all processes reply immediately.
  - If some process is in state HELD, then that process will not reply until it is finished.
  - If some process is in state WANTED and has a smaller timestamp than the incoming request, it will queue the request until it is finished.
  - If two or more processes request entry at the same time, then whichever bears the lowest timestamp will be the first to collect N-1 replies.
Figure 12.4
Ricart and Agrawala’s algorithm

On initialization
\[
\text{state} := \text{RELEASED};
\]

To enter the section
\[
\begin{align*}
\text{state} & := \text{WANTED}; \\
\text{Multicast request to all processes; } & \quad \text{request processing deferred here} \\
T & := \text{request’s timestamp;} \\
\text{Wait until } & (\text{number of replies received} = (N - 1)); \\
\text{state} & := \text{HELD};
\end{align*}
\]

On receipt of a request \(<T_i, p_i>\) at \(p_j\) \((i \neq j)\)
\[
\begin{align*}
\text{if } (\text{state} = \text{HELD} & \text{ or } (\text{state} = \text{WANTED} \text{ and } (T, p_j) < (T_i, p_i))) \\
\text{then} & \quad \text{queue request from } p_i \text{ without replying;} \\
\text{else} & \quad \text{reply immediately to } p_i; \\
\text{end if}
\end{align*}
\]

To exit the critical section
\[
\begin{align*}
\text{state} & := \text{RELEASED}; \\
\text{reply to any queued requests;}
\end{align*}
\]
P1 and P2 request CS concurrently. The timestamp of P1 is 41 and for P2 is 34. When P3 receives their requests, it replies immediately. When P2 receives P1’s request, it finds its own request has the lower timestamp, and so does not reply, holding P1 request in queue. However, P1 will reply. P2 will enter CS. After P2 finishes, P2 reply P1 and P1 will enter CS.

- Granting entry takes 2(N-1) messages, N-1 to multicast request and N-1 replies. Bandwidth consumption is high.
- Client delay is again 1 round trip time
- Synchronization delay is one message transmission time.
Maekawa’s voting algorithm

- It is not necessary for all of its peers to grant access. Only need to obtain permission to enter from subsets of their peers, as long as the subsets used by any two processes overlap.
- Think of processes as voting for one another to enter the CS. A candidate process must collect sufficient votes to enter.
- Processes in the intersection of two sets of voters ensure the safety property ME1 by casting their votes for only one candidate.
Maekawa’s voting algorithm

- A voting set \( V_i \) associated with each process \( p_i \).
- There is at least one common member of any two voting sets, the size of all voting set are the same size to be fair.
- The optimal solution to minimizes \( K \) is \( K \approx \sqrt{N} \) and \( M = K \).

\[
V_i \subseteq \{ p_1, p_2, \ldots, p_N \}
\]

such that for all \( i, j = 1, 2, \ldots, N \):

\[
p_i \in V_i
\]

\[
V_i \cap V_j \neq \emptyset
\]

\[
|V_i| = K
\]

Each process is contained in \( M \) of the voting set \( V_i \).
Figure 12.6
Maekawa’s algorithm – part 1

On initialization
state := RELEASED;
voted := FALSE;

For pi to enter the critical section
state := WANTED;
Multicast request to all processes in Vi;
Wait until (number of replies received = K);
state := HELD;

On receipt of a request from pi at pj
if (state = HELD or voted = TRUE)
then
queue request from pi without replying;
else
send reply to pi;
voted := TRUE;
end if

For pi to exit the critical section
state := RELEASED;
Multicast release to all processes in Vi;

On receipt of a release from pi at pj
if (queue of requests is non-empty)
then
remove head of queue – from pk, say;
    send reply to pk;
voted := TRUE;
else
    voted := FALSE;
end if
Maekawa’s algorithm

- ME1 is met. If two processes can enter CS at the same time, the processes in the intersection of two voting sets would have to vote for both. The algorithm will only allow a process to make at most one vote between successive receipts off a release message.

- Deadlock prone. For example, p1, p2 and p3 with V1={p1,p2}, V2={p2, p3}, V3={p3,p1}. If three processes concurrently request entry to the CS, then it is possible for p1 to reply to itself and hold off p2; for p2 rely to itself and hold off p3; for p3 to reply to itself and hold off p1. Each process has received one out of two replies, and none can proceed.

- **Bandwidth** utilization is $2\sqrt{N}$ messages per entry to CS and $\sqrt{N}$ per exit.

- **Client delay** is the same as Ricart and Agrawala’s algorithm, one round-trip time.

- **Synchronization delay** is one round-trip time.
Fault tolerance

• What happens when messages are lost?
• What happens when a process crashes?

• None of the algorithm that we have described would tolerate the loss of messages if the channels were unreliable.

  ▾ The ring-based algorithm cannot tolerate any single process crash failure.
  ▾ Maekawa’s algorithm can tolerate some process crash failures: if a crashed process is not in a voting set that is required.
  ▾ The central server algorithm can tolerate the crash failure of a client process that neither holds nor has requested the token.
  ▾ The Ricart and Agrawala algorithm as we have described it can be adapted to tolerate the crash failure of such a process by taking it to grant all requests implicitly.
Elections

*Algorithm to choose a unique process to play a particular role is called an election algorithm. E.g. central server for mutual exclusion, one process will be elected as the server. Everybody must agree. If the server wishes to retire, then another election is required to choose a replacement.*

*Requirements:*

- **E1(safety):** a participant $p_i$ has $elected_i = \perp$ or $elected_i = P$
  
  Where $P$ is chosen as the non-crashed process at the end of run with the largest identifier. (concurrent elections possible.)

- **E2(liveness):** All processes $P_i$ participate in election process and eventually set $elected_i \neq \perp$ or crash
A ring based election algorithm

- All processes arranged in a logical ring.
- Each process has a communication channel to the next process.
- All messages are sent clockwise around the ring.
- Assume that no failures occur, and system is asynchronous.
- Goal is to elect a **single** process coordinator which has the largest identifier.
Figure 12.7
A ring-based election in progress

1. Initially, every process is marked as non-participant. Any process can begin an election.
2. The starting process marks itself as participant and place its identifier in a message to its neighbour.
3. A process receives a message and compare it with its own. If the arrived identifier is larger, it passes on the message.
4. If arrived identifier is smaller and receiver is not a participant, substitute its own identifier in the message and forward it. It does not forward the message if it is already a participant.
5. On forwarding of any case, the process marks itself as a participant.
6. If the received identifier is that of the receiver itself, then this process’s identifier must be the greatest, and it becomes the coordinator.
7. The coordinator marks itself as non-participant set elected_i and sends an elected message to its neighbour enclosing its ID.
8. When a process receives elected message, marks itself as a non-participant, sets its variable elected_i and forwards the message.
A ring-based election in progress

- Note: The election was started by process 17.
- The highest process identifier encountered so far is 24.
- Participant processes are shown darkened.
- E1 is met. All identifiers are compared, since a process must receive its own ID back before sending an elected message.
- E2 is also met due to the guaranteed traversals of the ring.
- Tolerate no failure makes ring algorithm of limited practical use.

If only a single process starts an election, the worst-performance case is then the anti-clockwise neighbour has the highest identifier. A total of N-1 messages is used to reach this neighbour. Then further N messages are required to announce its election. The elected message is sent N times. Making **3N-1 messages in all.**

**Turnaround time** is also 3N-1 sequential message transmission time.
The bully algorithm

- Allows process to crash during an election, although it assumes the message delivery between processes is reliable.
- Assume system is synchronous to use timeouts to detect a process failure.
- Assume each process knows which processes have higher identifiers and that it can communicate with all such processes.
- Three types of messages:
  - **Election** is sent to announce an election message. A process begins an election when it notices, through timeouts, that the coordinator has failed.\[ T = 2T_{\text{trans}} + T_{\text{process}} \] From the time of sending
  - **Answer** is sent in response to an election message.
  - **Coordinator** is sent to announce the identity of the elected process.
Figure 12.8
The bully algorithm

1. The process begins an election by sending an election message to these processes that have higher IDs and awaits an answer in response. If none arrives within time $T$, the process considers itself the coordinator and sends coordinator message to all processes with lower identifiers. Otherwise, it waits a further time $T'$ for coordinator message to arrive. If none, begins another election.

2. If a process receives a coordinator message, it sets its variable elected_i to be the coordinator ID.

3. If a process receives an election message, it sends back an answer message and begins another election unless it has begun one already.

E1 may be broken if timeout is not accurate or replacement. (suppose P3 crashes and replaced by another process. P2 set P3 as coordinator and P1 set P2 as coordinator) E2 is clearly met by the assumption of reliable transmission.

The election of coordinator $p_2$, after the failure of $p_4$ and then $p_3$. 

Eventually....
The bully algorithm

- **Best case** the process with the second highest ID notices the coordinator’s failure. Then it can immediately elect itself and send N-2 coordinator messages.

- The bully algorithm requires $O(N^2)$ messages in the **worst case** - that is, when the process with the least ID first detects the coordinator’s failure. For then N-1 processes altogether begin election, each sending messages to processes with higher ID.
Consensus and Related Problems (agreement)

The problem is for processes to agree on a value after one or more of the processes has proposed what that value should be. (e.g. all controlling computers should agree upon whether let the spaceship proceed or abort after one computer proposes an action.

Assumptions: Communication is reliable but the processes may fail (arbitrary process failure as well as crash). Also specify that up to some number $f$ of the $N$ processes are faculty.
Consensus problem

Every process $p_i$ begins in the undecided state and propose a single value $v_i$, drawn from a set $D$. The processes communicate with one another, exchanging values. Each process then sets the value of a decision variable $d_i$. In doing so it enters the decided state, in which it may no longer change $d_i$.

Requirements:

- **Termination**: Eventually each correct process sets its decision variable.
- **Agreement**: The decision value of all correct processes is the same: if $p_i$ and $p_j$ are correct and have entered the decided state, then $d_i = d_j$.
- **Integrity**: if the correct processes all proposed the same value, then any correct process in the decided state has chosen that value. This condition can be loosen. For example, not necessarily all of them, may be some of them.

It will be straightforward to solve this problem if no process can fail by using **multicast** and **majority vote**.

Termination guaranteed by reliability of multicast, agreement and integrity guaranteed by the majority definition and each process has the same set of proposed value to evaluate.
Figure 12.17
Consensus for three processes

Consensus algorithm

\[ v_1 = \text{proceed} \]
\[ d_1 = \text{proceed} \]
\[ v_2 = \text{proceed} \]
\[ d_2 = \text{proceed} \]
\[ v_3 = \text{abort} \]

P_3 (crashes)
Byzantine general problem (proposed in 1982)

- Three or more generals are to agree to attack or to retreat. One, the commander, issues the order. The others, lieutenants to the commander, are to decide to attack or retreat.

- But one or more of the general may be treacherous—that is, faulty. If the commander is treacherous, he proposes attacking to one general and retreating to another. If a lieutenant is treacherous, he tells one of his peers that the commander told him to attack and another that they are to retreat.
Byzantine general problem and Interactive consistency

A. Byzantine general problem is different from consensus in that a distinguished process supplies a value that the others are to agree upon, instead of each of them proposing a value.

Requirements:
- Termination: eventually each correct process sets its decision variable.
- Agreement: the decision value of all correct processes is the same.
- Integrity: If the commander is correct, then all correct processes decide on the value that the commander proposed.

If the commander is correct, the integrity implies agreement; but the commander need not be correct.

B. Interactive consistency problem: Another variant of consensus, in which every process proposes a single value. Goal of this algorithm is for the correct processes to agree on a decision vector of values, one for each process.

Requirements: Termination: eventually each correct process sets its decision variable. Agreement: the decision vector of all correct processes is the same. Integrity: If pi is correct, then all correct processes decide on vi as the ith component of the vector.
Consensus in a synchronous system

- Basic multicast protocol assuming up to \( f \) of the \( N \) processes exhibit crash failures.
- Each correct process collects proposed values from the other processes. This algorithm proceeds in \( f+1 \) rounds, in each of which the correct processes Basic-multicast the values between themselves. At most \( f \) processes may crash, by assumption. At worst, all \( f \) crashes during the round, but the algorithm guarantees that at the end of the rounds all the correct processes that have survived have the same final set of values are in a position to agree.
Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization
$$\text{Values}_i^1 := \{v_i\}; \text{Values}_i^0 = \{\};$$

In round $r$ (1 ≤ $r$ ≤ $f + 1$)
$$B\text{-multicast}(g, \text{Values}_i^r - \text{Values}_i^{r-1}); // \text{Send only values that have not been sent}$$
$$\text{Values}_i^{r+1} := \text{Values}_i^r;$$
while (in round $r$)
$$\{$$

On $B\text{-deliver}(V_j)$ from some $p_j$
$$\text{Values}_i^{r+1} := \text{Values}_i^{r+1} \cup V_j;$$
$$\}$$

After $(f + 1)$ rounds
Assign $d_i = \text{minimum}(\text{Values}_i^{f+1});$

At most $f$ crashes can occur, and there are $f+1$ rounds. So we can compensate up to $f$ crashes.

Any algorithm to reach consensus despite up to $f$ crash failures requires at least $f+1$ rounds of message exchanges, no matter how it is constructed.
Figure 12.19
Three byzantine generals

Faulty processes are shown coloured

3:1:u: first number indicates source, the second number indicates Who says. From P3, P1 says u.
If solution exists, P2 bound to decide on v when commander is correct. If no solution can distinguish between correct and faulty commander, p2 must also choose the value sent by commander. By Symmetry, P3 should also choose commander, p2 does the same thing. But it contradicts with agreement. No solution is N<=3f. All because that a correct general can not tell which process is faulty. Digital signature can solve this problem.
First round: the commander sends a value to each of the lieutenants.

Second round: each of the lieutenants sends the value it received to its peers.

A lieutenant receives a value from the commander, plus N-2 values from its peers. Lieutenant just applies a simple majority function to the set of values it receives. The faulty process may omit to send a value. If timeouts, the receiver just set null as received value.