MODULE III

Time and Global States: Introduction, Clocks, Events and Process states, Synchronizing Physical clocks, Logical time and Logical clocks, Global states.

MODULE IV

Coordination and Agreement: Introduction, Distributed mutual exclusion, Elections, Multicast Communication, Consensus and Related problems.

Time and Global States: Introduction

- Time is an Important and interesting issue in distributes systems.
- One we can measure accurately.
- Can use as a metric.
- Example: e-commerce transaction timestamp for auditing and accountability purposes.
- Need time for maintaining consistency of databases.
- According to Einstein's theory of relativity a stationary observer on earth and an observer moving away from earth, if they observed an event at the same time, the event may "happen" at different times for them
- Relative order of two events can be reversed unless one caused the other.
- Thus the notion of physical time is problematic in distributed systems.
- We will examine synchronization of clocks using message passing;
- We will study logical clocks: vector clocks.
- We will also look at algorithms to capture global states of distributed systems as they execute.

Clocks, event and process states

- History (pi) = hi = $<e_i^0, e^1, e^2 \ldots>$
- $\sqsubset_i$ represents relation
- A processor clock is derived from a hardware clock:
- $C_i(t) = \alpha H_i(t) + \beta$
- May result in clock skew and drift
- Coordinated universal time: most accurate clock use atomic oscillators with very low drift rates of 1 in 10^{13}
- In use since 1967: standard second is defined as 9,192,631,770 periods of transitions between the two hyperfine levels of the ground state of Caesium-133 (Cs^{133})
- Days, hours, years are rooted in astronomical time.
- Japanese earthquake should have caused Earth to rotate a bit faster, shortening the length of the day by about 1.8 microseconds (a microsecond is one millionth of a second).
- Currents within earth's core also affect decreases and increases.
- Abbreviated UTC (is equivalent to Greenwich Mean Time (GMT)).
- This is based on atomic time (leap second is inserted, rarely leap second is deleted to keep up astronomical time).
- UTC signal are regularly synchronized and broadcast.

BCS515D,SKIT

- In USA the radio station WWV broadcasts time signals on several shortwave frequencies.
- Satellite sources for time include Global Positioning Systems (GPS).
- NIST is another source.



 Each node maintain a physical clock. However, they tend to drift even after an accurate initial setting. z

Skew: the difference between the readings of any two clocks.

z **Clock drift**: the crystal-based clock count time at different rates. Oscillator has different frequency. Drift rate is usually used to measure the change in the offset per unit of time. Ordinary quartz crystal clock, 1second per 11.6 days.

Synchronizing Physical clocks

 External synchronization: Ci is synchronized to a common standard.

z |S(t) –Ci(t)| <D, for i = 1,2,…N and for all real time t, namely clock Ci are accurate to within the bound D. S is standard time.

 Internal synchronization: Ci is synchronized with one another to a known degree of accuracy.

z |Ci(t) – Cj(t)| < D for i,j=1,2,…N, and for all real time t, namely, clocks Ci agree with each other within the bound D.

Synchronization in a synchronous system

 In a synchronous system, bounds exist for clock drift rate, transmission delay and time for computing of each step.

 One process sends the time t on it local clock to the other in a message m. The receiver should set its clock to t+Ttrans. It doesn't matter whether t is accurate or not

- Synchronous system: Ttrans could range from min to max. The uncertainty is u=(max-min). If receiver set clock to be t+min or t+max, the skew is as much as u. If receiver set the clock to be t+(min+max)/2, the skew is at most u/2.
- Asynchronous system: no upper bound max. only lower bound.

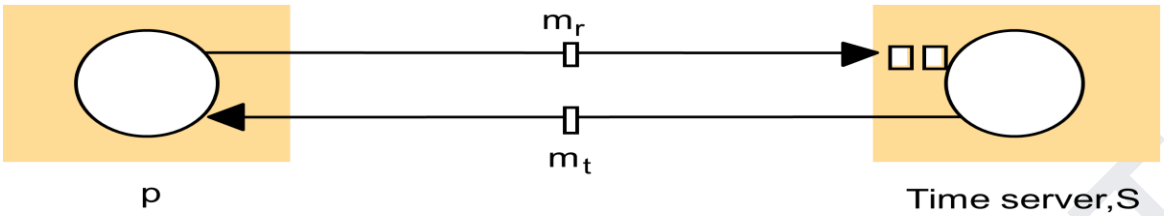
Cristian's method

- Cristian's method: Time server, connected to a device receiving signals from UTC. Upon request, the

BCS515D,SKIT

server S supplies the time t according to its clock.

- The algorithm is probabilistic and can achieve synchronization only if the observed round trip time are short compared with required accuracy.
- From p's point of view, the earliest time S could place the time in mt was min after p dispatch mr. The latest was min before mt arrived at p.



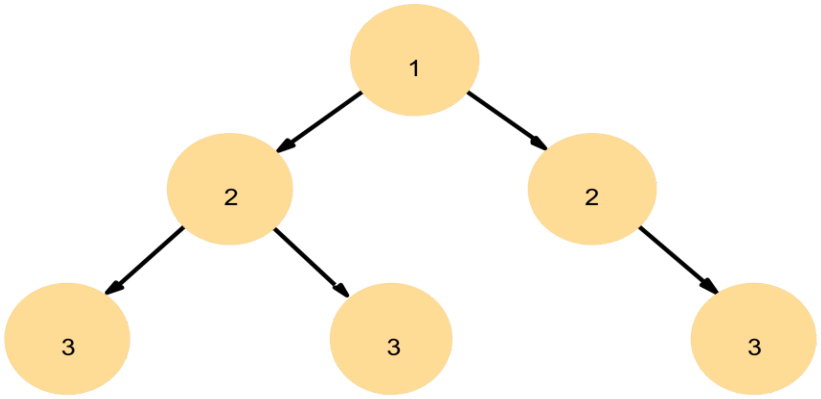
- The time of S by the time p receives the message mt is in the range of [t+min, t+Tround –min].
- P can measure the roundtrip time then p should set its time as (t + Tround/2) as a good estimation.
- The width of this range is (Tround -2min). So the accuracy is +-(Tround /2-min)
- Suffers from the problem associated with single server that single time server may fail.
- Cristian suggested to use a group of synchronized time servers. A client multicast is request to all servers and use only the first reply.
- A faulty time server that replies with spurious time values or an imposter time server with incorrect times.

Berkeley Algorithm

- Internal synchronization when developed for collections of computers running Berkeley UNIX.
- A coordinator is chosen to act as the master. It periodically polls the other computers whose clocks are to be synchronized, called slave. The salves send back their clock values to it. The master estimate their local clock times by observing the round-trip time similar to Cristian's method. It averages the values obtained including its own.
- Instead of sending the updated current time back to other computers, which further introduce uncertainty of message transmission, the master sends the amount by which each individual slave's clock should adjust.
- The master takes a fault-tolerant average, namely a subset of clocks is chosen that do not differ from one another by more than a specified bound.
- The algorithm eliminates readings from faulty clocks. Such clocks could have a adverse effect if an ordinary average was taken.

The Network Time Protocol

- Cristian's method and Berkeley algorithm are primarily for Intranets. The Network Time Protocol(NTP) defines a time service to distribute time information over the Internet.
  - Clients across the Internet to be synchronized accurately to UTC. Statistical techniques
  - Reliable service that can survive lengthy losses of connectivity. Redundant servers and redundant paths between servers.
  - Clients resynchronized sufficiently frequently to offset the rates of drift.
  - Protection against interference with time services. Authentication technique from claimed trusted



Note: Arrows denote synchronization control, numbers denote strata.

sources.

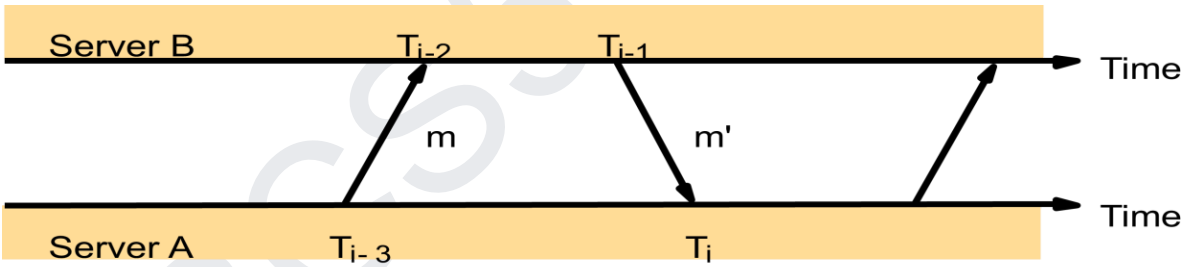
Hierarchical structure called synchronization subnet

- Primary server: connected directly to a time source.
- Secondary servers are synchronized with primary server.
- Third servers are synchronized with secondary servers.

Such subnet can reconfigure as servers become unreachable or failures occur.

NTP servers synchronize in one of three modes:

1. Multicast mode: for high-speed LAN. One or more servers periodically multicasts the time to servers connected by LAN, which set their times assuming small delay. Achieve low accuracy.
2. Procedure call: similar to Cristian's algorithm. One server receives request, replying with its timestamp. Higher accuracy than multicast or multicast is not supported.
3. Symmetric mode: used by servers that supply time in LAN and by higher level of synchronization subnet. Highest accuracy. A pair of servers operating in symmetric mode exchange messages bearing timing information.



Each message bears timestamps of recent message events: the local times when the previous NTP message between the pair was sent and received, and the local time when the current message was transmitted. The recipient of the NTP message notes the local time when it receives the message.

Logical time and Logical clocks

- In single process, events are ordered by local physical time. Since we cannot synchronize physical clocks perfectly across a distributed system, we cannot use physical time to find out the order of any arbitrary pair of events.
- We will use logical time to order events happened at different nodes. Two simple points:

- If two events occurred at the same process, then they occurred in the order in which pi observes them
- Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.

- Lamport (1978) called the partial ordering by generalizing these two relationships the happened-before relation.

$$HB1: If \exists\ process\ p_i : e \rightarrow e', then\ e \rightarrow e'$$

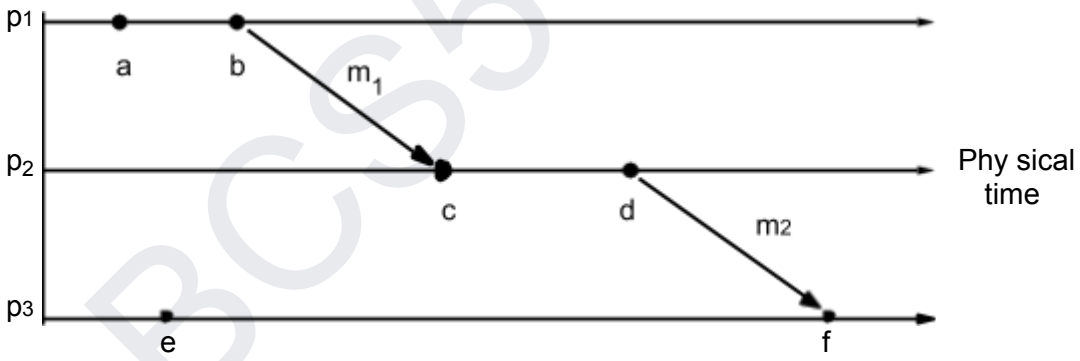
$$HB2 : For\ any\ message\ m, send(m) \rightarrow receive(m)$$

$$HB3 : If\ e, e', and\ e''\ are\ \ and$$

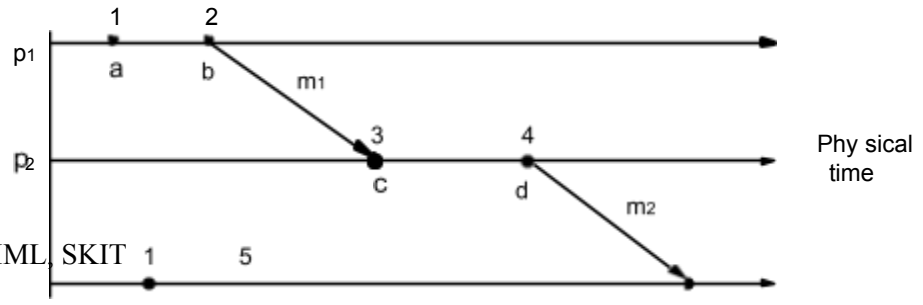
$$events, such\ that\ e \rightarrow e'$$

$$e' \rightarrow e'', then\ e \rightarrow e''$$

Events occurred at three processes



Lamport timestamps for the events



BCS515D,SKIT

p3

e f

$a \rightarrow_1 b$ and $c \rightarrow_2 d$

$b \rightarrow c$ and $d \rightarrow f$

combing them, $a \rightarrow f$

$a \not\rightarrow e$ and $e \not\rightarrow a$

concurrent $a \| e$

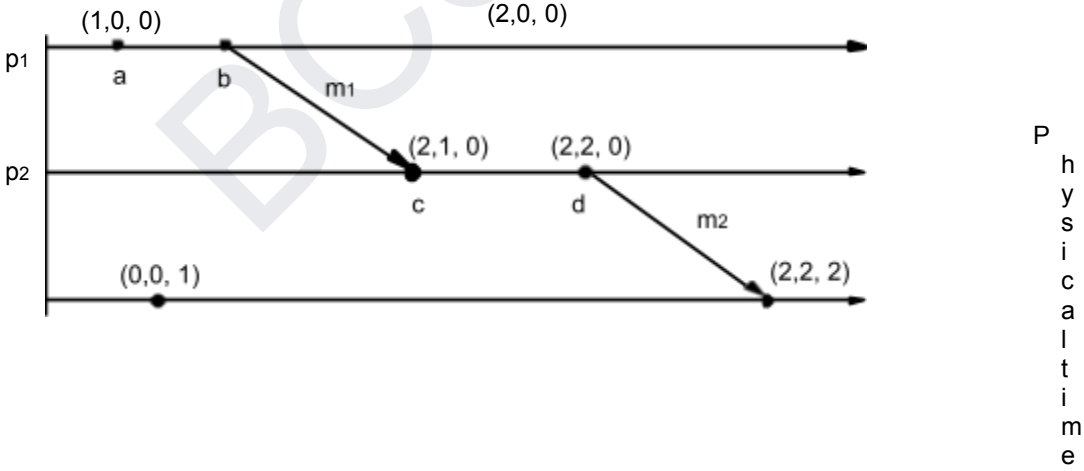
Logical Clocks

- Lamport invented a logical clock Li, which is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process pi keeps its own logical clock.
- LC1: Li is incremented before each event is issued at process pi: Li = Li +1
- LC2: a) Pi sends a message m, it piggybacks on m the value t = Li
  b) On receiving (m,t), a process pj computes Lj=max(Lj,t) and then applies LC1 before timestamping the event receive(m).
- It can be easily shown that:
- If e->e' then L(e) < L(e').
- However, the converse is not true. If L(e) < L(e'), then we cannot infer that e->e'. E.g b and e
- L(b)>L(e) but b||e
- How to solve this problem?

Vector Clock

- Lamport's clock: L(e)<L(e') we cannot conclude that e->e'.
- Vector clock to overcome the above problem.
- N processes is an array of N integers. Each process keeps its own vector clock Vi, which it uses to timestamp local events.

- VC1: initially, Vi[j] = 0, for i,j = 1,2…N
- VC2: just before pi timestamps an event, it sets Vi[i] = vi[i]+1
- VC3: pi includes the value t= Vi in every message it sends
- VC4: when pi receives a timestamp t in a message, it sets Vi[j]=max(Vi[j], t[j])for j =1,2…,N. Merge operation.



BCS515D, SKIT

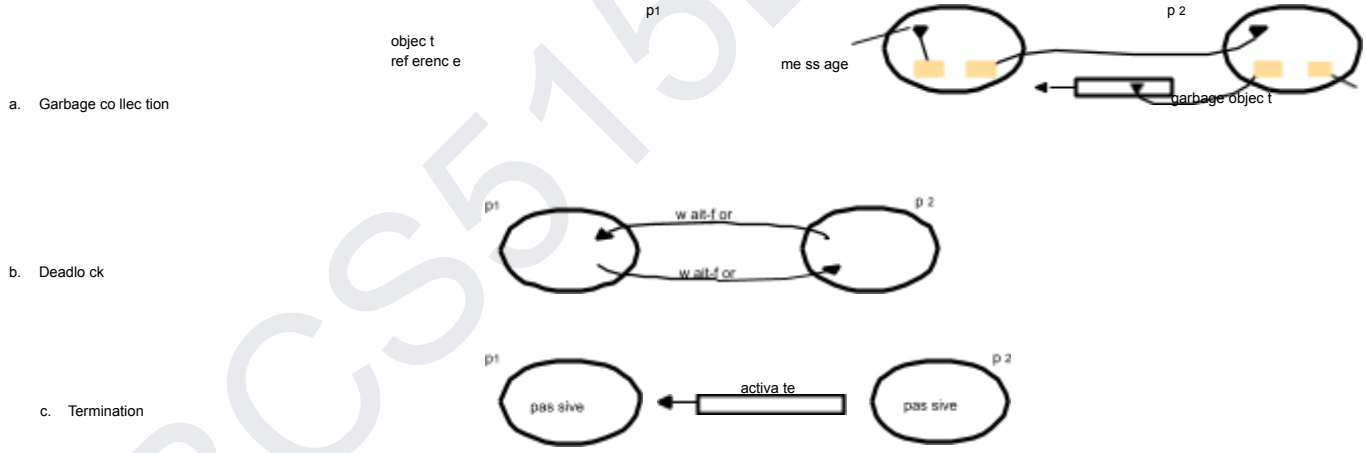
p3

e f

- To compare vector timestamps, we need to compare each bit. Concurrent events cannot find a relationship.
- Drawback compared with Lamport time, taking up an amount of storage and message payload proportional to N.

Global states

- We want to find out whether a particular property is true of a distributed system as it executes.
- We will see three examples:
  - Distributed garbage collection: if there are no longer any reference to objects anywhere in the distributed system, the memory taken up by the objects should be reclaimed.
  - Distributed deadlock detection: when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this "wait-for" relationship.
  - Distributed termination detection: detect if a distributed algorithm has terminated. It seems that we only need to test whether each process has halted. However, it is not true. E.g. two processes and each of which may request values from the other. It can be either in passive or active state. Passive means it is not engaged in any activity but is prepared to respond. Two processes may both in passive states. At the same time, there is a message in on the way from P2 to P1, after P1 receives it, it will become active again. So the algorithm has not terminated.



Global States and consistent cuts

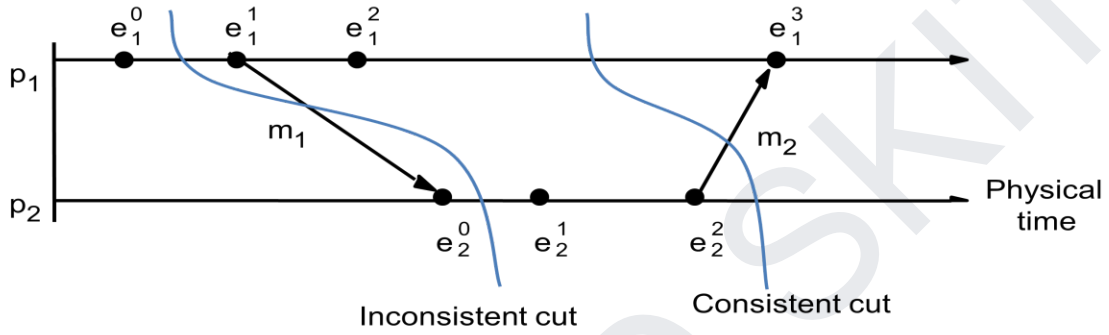
- It is possible to observe the succession of states of an individual process, but the question of how to ascertain a global state of the system – the state of the collection of processes is much harder.
- The essential problem is the absence of global time. If we had perfectly synchronized clocks at which processes would record its state, we can assemble the global state of the system from local states of all processes at the same time.
- The question is: can we assemble the global state of the system from local states recorded at different real times?
- The answer is "YES".

BCS515D SKIT

$history(p_i) = h_i = < e_i^0, e_i^1, e_i^2, ... >$

$finite\ prefix\ of\ history : h_i^k = < e_i^0, e_i^1, e_i^2, ... e_i^k >$

- A series of events occurs at each process. Each event is either an internal action of the process (variables updates) or it is the sending or receipt of a message over the channel.
- S^k_i is the state of process Pi before kth event occurs, so S^0_i is the initial state of Pi.
- Thus the global state corresponds to initial prefixes of the individual process histories.
- A cut of the system's execution is a subset of its global history that is a union of prefixes of process histories $C = h^{c_1} \cup h^{c_2} \cup ... \cup h^{c_N}$
- The state of each process is in the state after the last event occurs in its own cut. The set of last



events from all processes are called frontier of the cut.

- Inconsistent cut: since P2 contains receiving of m1, but at P1 it does not include sending of that message. This cut shows the an effect without a cause. We will never reach a global state that corresponds to process state at the frontier by actual execution under this cut.
- **Consistent cut**: it includes both the sending and receipt of m1. It includes the sending but not the receipt of m2. It is still consistent with actual execution.
- A cut C is **consistent** if, for each event it contains, it also contains all the events that happened-before that event.

$$for\ all\ events\ e \in C, f \rightarrow e \Rightarrow f \in C$$

- A consistent global state is one that corresponds to a consistent cut.
- A run is a total ordering of all the events in a global history that is consistent with each local history's ordering.
- A linearization or consistent run is an ordering of the events in a global history that is consistent with this happened-before relation.

Global state predicate

- Global state predicate is a function that maps from the set of global states of processes n the system to true or false.
- Stable characteristics associated with object being garbage, deadlocked or terminated: once the system enters a state in which the predicate is True. It remains True in all future states reachable from that state.
- Safety (evaluates to deadlocked false for all states reachable from S0)
- Liveness (evaluate to reaching termination true for some of the states reachable from S0)

BOSE D, SKIT

Chandy and Lamport's 'snapshot' algorithm

- Chandy and Lamport(1985) describe a "snapshot" algorithm for determining global states of distributed system.
- Record a set of process and channel states for a set of processes Pi such that even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent.
- The algorithm records state locally at processes without giving a method for gathering the global state.
- Assumptions:
  1. Neither channels nor processes fail; communication is reliable so that every message sent is eventually received intact, exactly once;
  2. Channel are unidirectional either incoming or outgoing and provide FIFO order message delivery;
  3. The graph of processes and channels is strongly connected (there is a path between any two processes).
  4. Any process may initiate a global snapshot at any time.
  5. The processes may continue their normal execution and send and receive normal massages while the snapshot takes place.
- Each process records its own state and also for each incoming channel a set of messages sent to it.
- Allow us to record process states at different times but to account for the differential between process states in terms of message transmitted but not yet received.
- If process pi has sent a message m to process pj, but pj has not received it, then we account for m as belong to the state of the channel between them.
- Use of special marker message. It has a dual role, as a prompt for the receiver to save its own state if it has not done so; and as a means of determining which messages to include in the channel state.
- **********

On p_i's receipt of a *marker* message over channel c:
  if (p_i has not yet recorded its state) it
      records its process state now;
      records the state of c as the empty set;
      turns on recording of messages arriving over other incoming channels;
  else
      p_i records the state of c as the set of messages it has received over c since it saved its state.
  end if

Marker sending rule for process p_i
  After p_i has recorded its state, for each outgoing channel c:
    p_i sends one marker message over c
    (before it sends any other message over c).

MODULE-4

COORDINATION AND AGREEMENT: Introduction, Distributed mutual exclusion, Elections, Coordination and agreement in group communication, Consensus and related problems.

Coordination and Agreement: Introduction

- 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 process **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
    - resrouceAccess() //access shared resoruces
    - 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

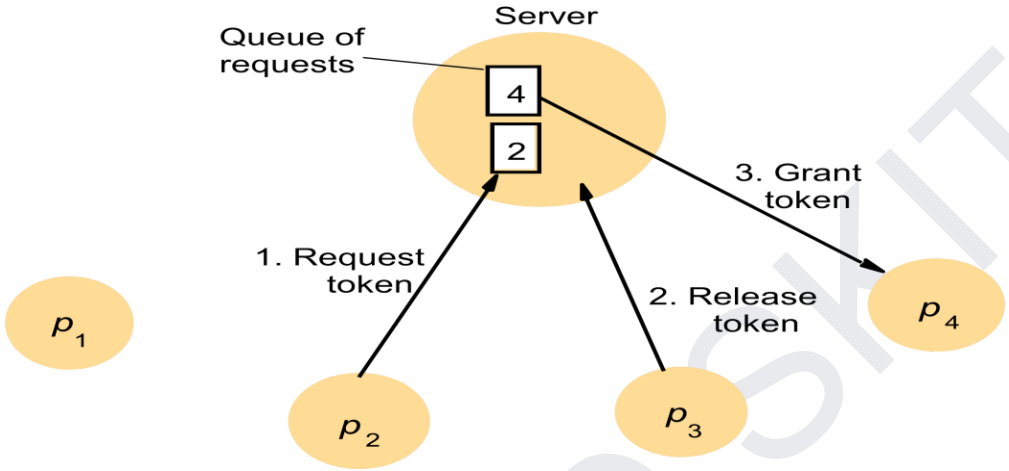
- Bandwidth consumption, which is proportional to the number of messages sent in each entry and exit operations.
- The client delay incurred by a process at each entry and exit operation.
- throughput of the system. Rate at which the collection of processes as a whole can access the critical section. 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.

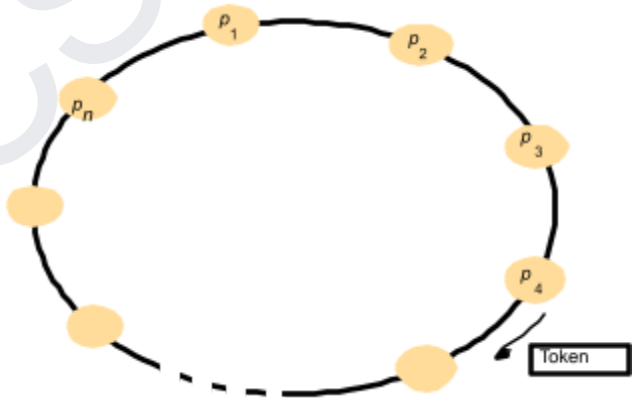
BCS515D,SKIT

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



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.



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

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 pi keeps a Lamport clock. Message requesting entry are of the form<T, pi>.
- 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.

Ricart and Agrawala's algorithm

```
On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast request to all processes;      } request processing deferred
    T := request's timestamp;
    Wait until (number of replies received = (N – 1));
    state := HELD;

On receipt of a request <Ti, pi> at pj (i ≠ j)
    if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
    then
        queue request from pi without replying;
    else
        reply immediately to pi;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

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
- A voting set Vi associated with each process pi.
- 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~sqrt(N) and M=K.

$$V_i \subseteq \{p_1, p_2, ..., p_N\}$$

$$such\ that\ \ for\ all\ i,\ j = 1,2,...N:$$

$$p_i \in V_i$$

Maekawa's algorithm

On initialization
  state := RELEASED;
  voted := FALSE;
For p_i to enter the critical section
  state := WANTED;
  Multicast *request* to all processes in V_i;
  Wait until (number of replies received = *K*);
  state := HELD;
On receipt of a request from p_i at p_j
  if (*state* = HELD *or voted* = TRUE)
  then
    queue *request* from p_i without replying;
  else
    send *reply* to p_i;
    voted := TRUE;
  end if

For p_i to exit the critical section
  state := RELEASED;
  Multicast *release* to all processes in V_i;
On receipt of a release from p_i at p_j
  if (queue of requests is non-empty)
  then
    remove head of queue – from p_k, say;
    send *reply* to p_k;
    voted := TRUE;
  else
    voted := FALSE;
  end if

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 pi 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 Pi participate in election process and eventually set $elected_i \neq \perp$ *or crash*

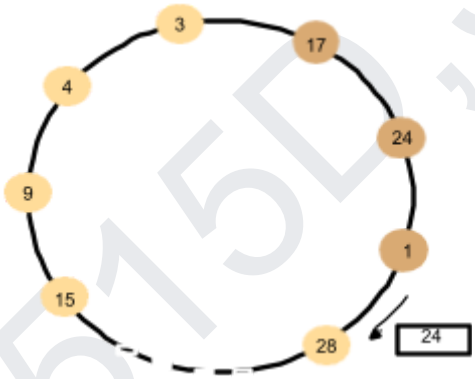
A ring based election algorithm

- 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.

  - **Initially**, every process is marked as non-participant. Any process can begin an election.
  - The **starting** process marks itself as participant and place its identifier in a message to its neighbour.

- A process receives a message and **compare** it with its own. If the arrived identifier is **larger**, it passes on the message.
- If arrived identifier is **smaller** and receiver is not a participant, substitute its own identifier in the message and forward if. It does not forward the message if it is already a participant.

- On forwarding of any case, the process marks itself as a participant.
- If the received identifier is that of the receiver itself, then this process' s identifier must be the greatest, and it becomes the **coordinator**.
- The coordinator marks itself as non-participant set elected_i and sends an **elected** message to its neighbour enclosing its ID.
- When a process receives elected message, marks itself as a non-participant, sets its variable elected_i and forwards the message.



- 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

BCS515D,SKIT

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=2Ttrans+Tprocess 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.

1. The process begins a election by sending an election message to these processes that have a higher ID 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 send 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.

- 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)

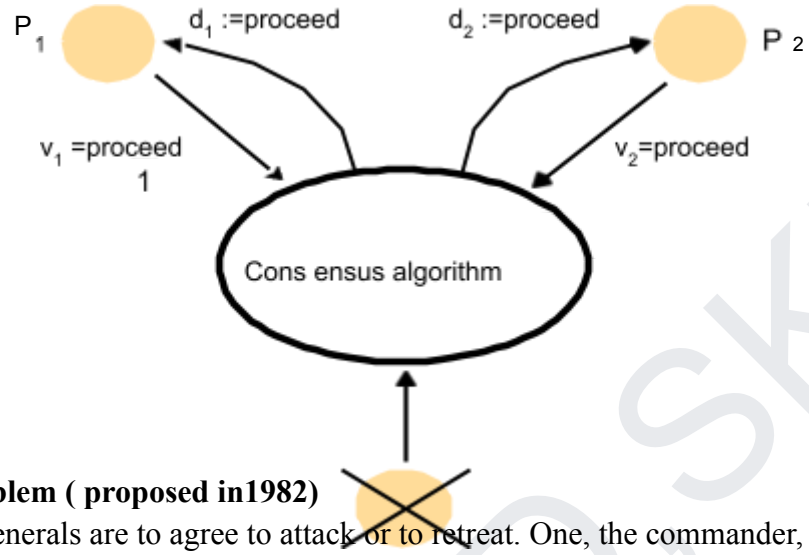
- The problem is for processes to agree on a value after one or more of the processes has proposed what that value should be. (e.g. all controlling computers should agree upon whether let the spaceship proceed or abort after one computer proposes an action.)
- Assumptions: Communication is reliable but 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.
- Every process pi begins in the undecided state and propose a single value vi, drawn from a set D. The processes communicate with one another, exchanging values. Each process then sets the value of a decision variable di. In doing so it enters the decided state, in which it may no longer change di.
- Requirements:
    - Termination: Eventually each correct process sets its decision variable.

    - Agreement: The decision value of all correct processes is the same: if pi and pj are correct and have entered the decided state, then di=dj

BCS515D,SKIT

- 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. .



Byzantine general problem (proposed in1982)

- 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.
- 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 it 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.

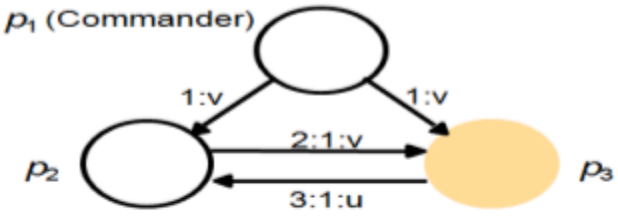
Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

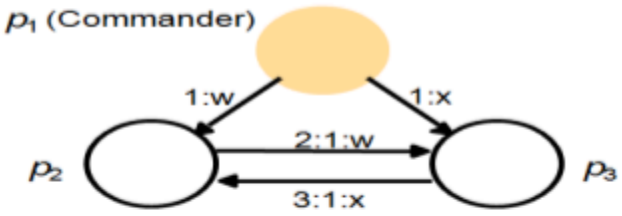
On initialization
$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \le r \le f + 1)$
B-multicast$(g, Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent
$Values_i^{r+1} := Values_i^r$;
while (in round *r*)
{
On B-deliver(V_j) *from some* p_j
$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;
}

After $(f + 1)$ *rounds*
Assign d_i = *minimum*$(Values_i^{f+1})$;

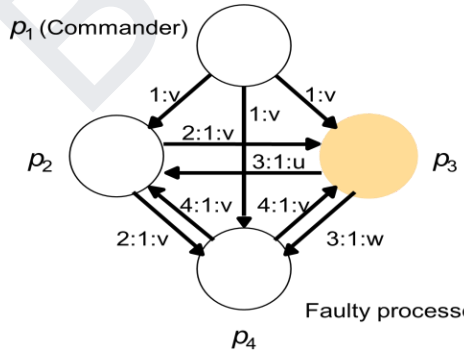
Three byzantine generals

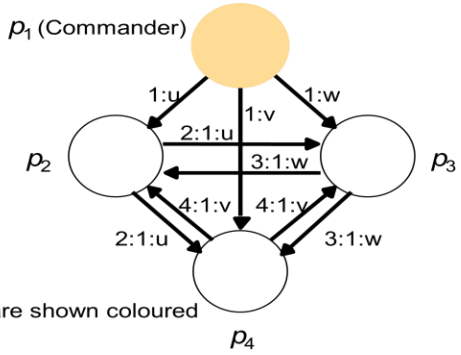




Faulty processes are shown coloured

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.





Faulty processes are shown coloured

Multicast Communication

- Group (multicast) communication: for each of a group of processes to receive copies of the messages sent to the group, often with delivery guarantees
  - The set of messages that every process of the group should receive
  - On the delivery ordering across the group members
- Challenges
  - Efficiency concerns include minimizing overhead activities and increasing throughput and bandwidth utilization
  - Delivery guarantees ensure that operations are completed
- Types of group
  - Static or dynamic: whether joining or leaving is considered
  - Closed or open
    - A group is said to be closed if only members of the group can multicast to it. A process in a closed group sends to itself any messages to the group
    - A group is open if processes outside the group can send to it
- Simple basic multicasting **(B-multicast)** is sending a message to every process that is a member of a defined group
  - B-multicast(g, m) for each process p Î group g, send(p, message m)
  - On receive(m) at p: B-deliver(m) at p
- **Reliable multicasting (R-multicast)** requires these properties
  - Integrity: a correct process sends a message to only a member of the group and does it only once
  - Validity: if a correct processsends a message, it will eventually be delivered.
  - Agreement: if a message is delivered to a correct process, all other correct processes in the group will deliver it

On initialization
Received := {};

For process p to R-multicast message m to group g
B-multicast(*g*, *m*); // $p \in g$ is included as a destination

On B-deliver(*m*) *at process q with g = group*(*m*)
if ($m \notin$ *Received*)
then
Received := *Received* $\cup$ {*m*};
if ($q \neq p$) *then B-multicast*(*g*, *m*); *end if*
R-deliver m;
end if

- Implementing reliable R-multicast over B-multicast
  - When a message is delivered, the receiving process multicasts it

- Duplicate messages are identified (possible by a sequence number) and not delivered

Types of message ordering

Three types of message ordering

- ***FIFO (First-in, first-out) ordering***: if a correct process delivers a message before another, every correct process will deliver the first message before the other
- ***Casual ordering*****:** any correct process that delivers the second message will deliver the previous message first
- ***Total ordering*****:** if a correct process delivers a message before another, any other correct process that delivers the second message will deliver the first message first

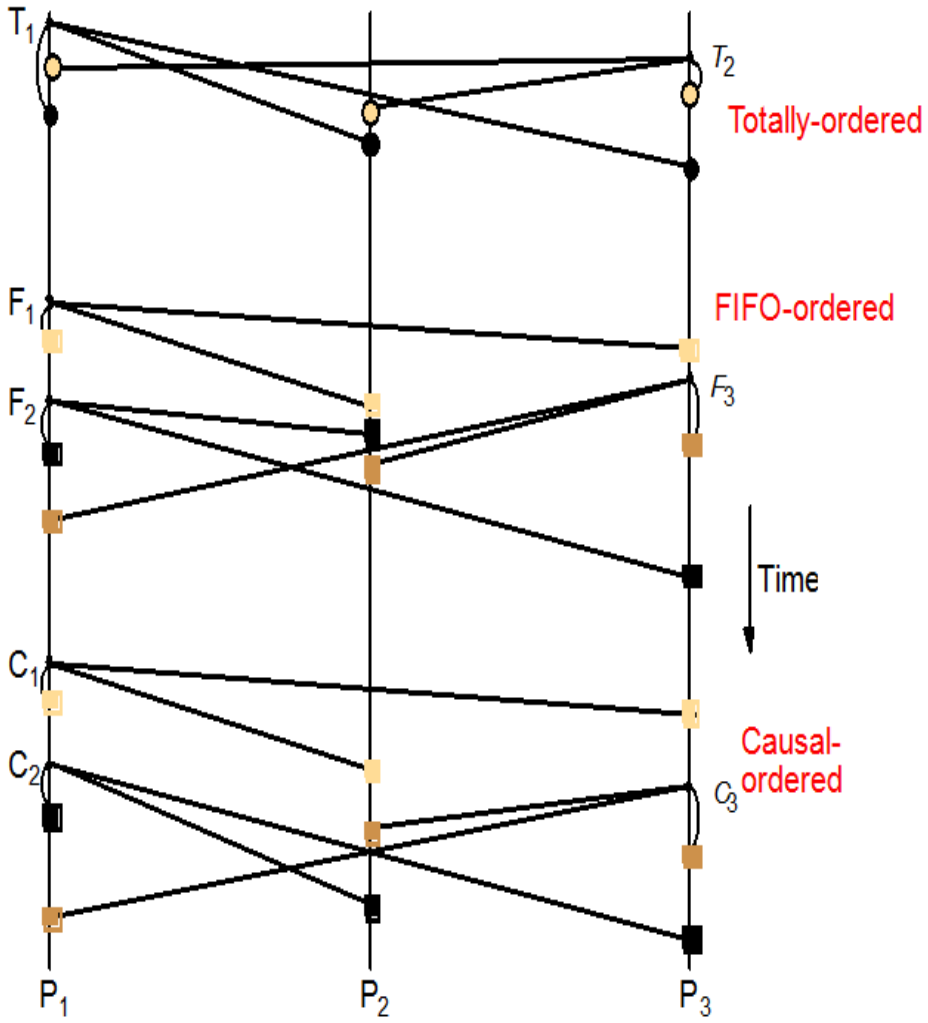
• Note that

- FIFO ordering and casual ordering are only partial orders
- Not all messages are sent by the same sending process
- Some multicasts are concurrent, not able to be ordered by happened- before
- Total order demands consistency, but not a particular order –

BCS515D, SKIT
CSE DEPARTMENT (MRCET)

- **Totally ordered** messages T_1 and T_2.
- **FIFO-related** messages F_1 and F_2.
- **Causally-related** messages C_1 and C_3

- Causal ordering implies FIFO ordering
- Total ordering does not imply causal ordering.
- Causal ordering does not imply total ordering.
- Hybrid mode: causal-total ordering, FIFO-total ordering.



Multicast Communication

- Group (multicast) communication: for each of a group of processes to receive copies of the messages sent to the group, often with delivery guarantees
  - The set of messages that every process of the group should receive
  - On the delivery ordering across the group members
- Challenges
  - Efficiency concerns include minimizing overhead activities and increasing throughput and bandwidth utilization
  - Delivery guarantees ensure that operations are completed
- Types of group
  - Static or dynamic: whether joining or leaving is considered

  - Closed or open
    - A group is said to be closed if only members of the group can multicast to it. A process in a closed group sends to itself any messages to the group
    - A group is open if processes outside the group can send to it

- Simple basic multicasting **(B-multicast)** is sending a message to every process that is a member of a defined group
  - B-multicast(g, m) for each process p Î group g, send(p, message m)
  - On receive(m) at p: B-deliver(m) at p
- **Reliable multicasting (R-multicast)** requires these properties
  - Integrity: a correct process sends a message to only a member of the group and does it only once
  - Validity: if a correct processsends a message, it will eventually be delivered.
  - Agreement: if a message is delivered to a correct process, all other correct processes in the group will deliver it

On initialization
Received := {};

For process p to R-multicast message m to group g
B-multicast(*g*, *m*); // $p \in g$ is included as a destination

On B-deliver(*m*) *at process q with g = group*(*m*)
if ($m \notin$ *Received*)
then
Received := *Received* $\cup$ {*m*};
if ($q \neq p$) *then B-multicast*(*g*, *m*); *end if*
R-deliver m;
end if

- Implementing reliable R-multicast over B-multicast
  - When a message is delivered, the receiving process multicasts it
  - Duplicate messages are identified (possible by a sequence number) and not delivered

Types of message ordering

Three types of message ordering

- ***FIFO (First-in, first-out) ordering***: if a correct process delivers a message before another, every correct process will deliver the first message before the other
- ***Casual ordering*****:** any correct process that delivers the second message will deliver the previous message first
- ***Total ordering*****:** if a correct process delivers a message before another, any other correct process that

delivers the second message will deliver the first message first

- Note that
  - FIFO ordering and casual ordering are only partial orders
  - Not all messages are sent by the same sending process
  - Some multicasts are concurrent, not able to be ordered by happened- before
  - Total order demands consistency, but not a particular order –

- **Totally ordered** messages T_1 and T_2.
- **FIFO-related** messages F_1 and F_2.
- **Causally-related** messages C_1 and C_3

- Causal ordering implies FIFO ordering
- Total ordering does not imply causal ordering.
- Causal ordering does not imply total ordering.
- Hybrid mode: causal-total ordering, FIFO-total ordering.

