Cristian's algorithm: Syn nodes **with a given bound**: Exter. syn. 1. Each client asks time at every R interval. 2. Client sets time to $T_S + {}^{RTT}/_{2}$. $RTT = T_2 - T_1$. **Ass sym latency**. Acc is $\pm({}^{RTT}/_{2} - Min)$. Min = minimum time. NTP based in this concept. **Berkeley algorithm:** Keep clks in sync **within a bound**: Intern. syn. 1. Master polls slaves clks at every R (adapting them). 2. Master calculates average (Clks outside bound are discarded). 3. Master sends **adjustment** to be made to each local clk. **Lamport's logical clocks:** Each process Pi mantains a local counter Ci. Ci is used to attach a timestampt to each event. Ci is adjusted according to the following rules: 1. When an event happens at Pi, it increases Ci by one. 2. When Pi sends a message m to Pj, sets ts(m) = Ci. 3. When Pj receives m, sets Cj = max(Cj, ts(m)), and then increases by one. Lamport's clock define a **partial order** that is consistent with the happened-before relation. **Vector clocks:** Each process has array of vector clks. Each pos known clk of each process. VC adjusted: 1. When Pi sends a message ++ to its position. 2. When Pj receives a message updates **each** position taking the max. **Detects causality**. **Centralized algorithm:** 1. To enter the CS, a process sends a **Request** to the coordinator and **awaits permission**. 2. If no other process is in the CS, coordinator grants access (**OK**), otherwise queues the request. 3. Send a Release to the coordinator to exit the CS. 4. Coordinator removes the oldest process in the queue and grants access to it (**OK**). Problems: 1. What if coordinator crashes? A process can't distinguish between lock or crash 2. Coordinator can be bttlneck. Bandwidth: 3 ({request + OK} + release). Client delay: round-trip (request + OK). Synch. Delay: round-trip (release + OK). Safety: Yes. Liveness: Yes. Happened-before ordering: No. (Assuming process do not fail) **Lin's voting algorithm:** Decentralized algorithm with **N coordinators.** Using voting, gain **quorum. 1.** To enter critical section process sends Request to **all** the **coordinators** (including Lamport's clock and its process ID). **2.** A coordinator grants vote if it has not voted yet, otherwise it queues request (ordering by logical time), in any case it sends a Response message to requester including its current vote. **3.** Requester analyses the votes in the responses: **a)** If the requester obtains the **majority** vote it can enter the critical section. **b)** If someone else gets majority does nothing. **c)** If nobody gets majority the requester: **OptA**: Releases its votes and backs off for a random period (Might cause Starvation). **OptB**: Sends a Yield message (release+request) to all the coordinators that voted for it (No Starvation). **4.** On receiving a Yield message, a coordinator: Removes its vote, queues the request, gets the head of the queue and votes for it, Sends a Response message to the voted process, if vote has changed, it notifies also the former process. **5.** To exit the critical section the requester sends a Release message to **all** coordinators. Each coordinator that voted for it: Removes its vote, Gets the head of the queue and votes for it, Sends Response message to the new voted process. The rest of coordinators simply dequeue the process. **Problems: 1.** Failure/recovery of coordinators, if a coordinator crashes and forgets previous vote => If there are f crashes and N-M+f >= M , the safety of the algorithm will be violated. **2.** Low efficiency if many Yield round are needed. Bandwidth (k Yield rounds): 3N + f(k) ({N request + N vote +$\sum_{i=1}^{k}(C_i\ yield + [C_i, 2C_i]\ vote)$} +N release) Client delay: round-trip. Synch. Delay: round-trip. Safety: Yes. Liveness: Yes (using Yield). Hapned-befre ordering: No **Ricart & Agrawala's algorithm:** Multicast with logical clocks. **1.** To enter the critical section, a process sends Request messages to **all** the other **processes** (including Lamport's clock and its process ID, process ID used to break ties). **2.** When the requester receives OK messages from **all** the processes, it can access the critical section. **2.** When a process receives a Request message: **a)** If it is not accessing the critical section and does not want to access sends back an OK message. **b)** If it is accessing now: no reply and queues the request. **c)** If it wants to access as well, but has not yet done so: If the incoming message has **lowe logical time:** sends back an OK message. Otherwise: no reply and queues the request. **4.** To exit the critical section, processes sends an OK message to each process in its queue. Problem: What if any process fails? The requester cannot distinguish between crash and denial. Solution: get permission from a majority. Bandwidth: 2(N-1) (N-1 request + N-1 OK). Client Delay: round-trip (N-1 request + N-1 OK). Synch. Delay: 1 (OK). Safety: Yes. Liveness: Yes. Happened-before ordering: Yes. **Maekawa's voting algorithm:** Get permission to access the CS by obtaining votes from a subset of the rest of processes Si. Unique voting set. All the voting sets have the same size K. Each process belongs to M sets. Intersection of any two voting sets is **non-empty.** Processes in the intersection ensure the **safety property** by voting for only one candidate. **1.** Pi Sends a Request message to all K members of Si. Pi needs OK from all of them to access the critical section. **2.** When a process in Si receives a Request message: If it is not accessing the critical section and has not already voted since it last received a Release message, sends an OK message immediately. Otherwise, It queues the request (in the order of its arrival) but does not yet reply. **3.** To exit the critical section Pi sends Release messages to all K processes in Si **4.** When a process in Si receives a Release message: It removes the head of its queue and sends an OK message (a vote) to it. **Problems: 1.** The algorithm is deadlock-prone (Liveness not guaranteed) **2.** Cannot tolerate failures within the required voting set, but can tolerate failures in other sets. Optimal: $K = M \approx \sqrt{N}$ Bandwidth: 3K ({K request + K OK} + K release). Client Delay: round-trip (K request + K OK). Synch. Delay: round-trip (release + OK). Safety: Yes. Liveness. No. Happened-before ordering: No. **Token ring algorithm:** Organize processes ina logical unidirectional ring (a process only knows its successor). A token message circulates around the ring. Only the process holding the token can enter the critical section. To exit the critical section or if the process does not want to enter, send the token to successor. **Problems: 1.** What if the token is lost? If the token is ever lost, it must be regenerated. Detecting token loss is difficult, requires coordination. **2.** What if a process crashes? Send the token to the next member down the line. Each process must know all the nodes in the ring. Bandwidth: 1 (all enter) to inf (none enter). Client Delay: 0 (token just received) to N-1 (token just departed). Synch delay: 1 (successor will access) to N-1 (predecessor will access). Safety: Yes. Liveness: Yes. Happened-before ordering: No. **Bully algorithm:** Assumptions: The system is **synchronous** (it is possible to use timeouts). Topology is a strongly connected graph. Message delivery between processes is reliable. Each process knows the ID of every other process, but not which ones are now up or down. Several processes may start elections concurrently. **1.** P sends an Election message to all the processes with **higher IDs** and awaits OK messages. **2.** If a process receives an Election message, it returns an Ok and starts another election, **unless it has begun one already 3.** If P receives an OK, it drops out the election and awaits a Coordinator message. P reinitiates the election if this message is not received. **4.** If P does not receive any OK before the timeout, it winds and sends a Coordinator message to the rest. It can violate the safety property if crashed process with the highest ID restarts during an ongoing election. **Chang and Robert's ring algorithm:** Processes are organized by ID in a logical unidirectional ring. Each process only knows its successor in the ring. Assumes that system is asynchronous. Multiple elections can be in progress. Redundant election messages are killed off. **1.** P sends an Election message (with its ID) to its successor, and becomes a participant. **2.** On receiving an Election, Q compares the ID in the message with its own ID: **a)** If the arrived ID is grater, Q forwards the message to its successor. **b)** If the arrived ID is smaller: If Q is nor a participant yet, Q replaces the ID in the message with its own ID and forwards it. If Q is already a participant, message is not forwarded. **c)** If the arrived ID is Q's ID, Q wins and sends a Coordinator message to its successor: Q becomes a participant on forwarding and Election. **3.** When Q gets a Coordinator message, it forwards it to successor (unless Q is the new coordinator) and becomes a non-participant. **Problems:** Liveness violated when process failure occurs during the election. Solution: Enhanced ring algorithm. **Enhanced ring algorithm: 1.** P sends an Election message (with its process ID) to its **closest alive** successor. Sequentially poll successors until one responds: Each process must know all nodes in the ring. **2.** At each step along the way, each process adds its ID to the list in the message. **3.** When the message gets back to the initiator, it elects as Coordinator the process with the highest ID and sends a Coordinator message with this ID. **4.** Again, each process adds its ID to the message **5.** Once Coordinator message gets back to initiator election is over, otherwise the election is reinitiated. U: Algorithm can support failures during the election. D: Redundant election messages are not killed off. **Comparison of election algorithms:** Bully algorithm: Worst case: initiator has the lowest ID $\theta(n^2)$ messages (Triggers N-1 elections). Best case: initiator has the highest ID: N-1 Coordinator messages, which can be sent in parallel. **Chang and Robets' ring algorithm:** Worst case: initiator succeeds the node with the highest ID: 3N-1 (2N-1 Election + N Coordinator) sequential messages. Best case: initiator has the highest ID: 2N (N election + N Coordinator) sequential messages. **Enhanced ring algorithm:** 2N (N election + N Coordinator) sequential messages always. None of the can tolerate network partitions, although some of them can tolerate failures to some extent. **Basic reliable multicast:** Simple solution assuming that processes **do not fail** and do not join/leave the group. Sender P assigns a sequence number Sp to each outgoing message (makes ez to spot a message is missing). P stores a copy of each outgoing message in a **history buffer.** P removes a message from the history buffer when everyone has ack receipt. Each process Q records the number of the **last message it has delivered** coming from any other process. When process Q receives a message from P: If the msgID is = to lastmsgID +1 then it delivers the message, if the msgID is > to lastmsgID +1 then it stores it in a hold-back queue and requests the restransmission of missing messages. Retransmission is also multicast message. If msgID is < to lastmsgID +1 then Q ahs already delivered the message before and thus it discards it. D: Poor scalability: too many ACKs (feedback implosion).

Scalable reliable multicast: Main idea: use sequence numbers but reduce the number of feedback messages to sender. Only missing messages are reported (NACK). NACKs are also multicast to all the group members. Each process waits a random delay prior to sending a NACK: If a process is about to NACK, this is suppressed as a result of the first multicast NACK. In this way, only one NACK will reach the sender. U: Better scalability. D: Setting timers to ensure only one NACK is hard. D: Sender should keep messages in the history buffer forever to guarantee all the retransmissions. In practice, messages are delated after some time. **Ordered multicast:** Due to the latency, messages might arrive in different order at different nodes. Common ordering requirements: **1.** FIFO ordering: messages from the same process delivered in the sent order by all processes. **2.** Causal ordering: happened-before-related messages delivered in that order by all processes. Causal ordering implies FIFO ordering. **3.** Total ordering: all messages delivered in the same order by all processes. Hybrid approaches such as FIFO + total ordering a causal + total ordering is also possible. Implementation hints: **FIFO ordering:** Using sequence numbers per sender. A message delivery is delayed (in a hold-back queue) until its sequence number is reached. **Total ordering:** Using sequence numbers per group. Two opt: **a).** Send messages to a sequencer, which multicasts them with numbering. A message delivery is delayed (in a hbq) until its number is reached. Sequencer is a single point of failure and a performance bottleneck. **b).** Processes jointly agree on sequence numbers. **1.** The sender multicast message m **2.** Each receiver replies with a proposed sequence numbers for message m (including its process ID) that is Pk=Max(Aj,Pj)+1 and places m in an ordered hold-back queue according to Pj. **3.** The sender selects the largest of all proposals, N, as the agreed number from m and multicasts it. **4.** Each receiver j updates Aj=Max(Aj, N), tags message m with N, and reorders the hold-back queue if needed. **5.** A message is delivered when it is at the front of the hbq and its number is agreed. **Causal ordering:** Using vector clk: A msg is delivered only if all causally preceding msgs have already been delivered. **1.** Pi increases VCi[i] only when sending a message **2.** If Pj receives message m form Pi, it postpones its delivery until the following conditions are met: VC(m)[i]=VCj[i] +1 (m is the next expected message from Pi). VC(m)[k]<=VCj[k] all k!=I (Pj has seen all the previous messages seen by Pi before m). **3.** Pj increases the corresponging clk. **Atomic multicast:** Solution for reliable multicasting in open groups (with faulty processes). Guarantee that a message is delivered to either all processes or none at all. A message is delivered only to the current non-faulty members of the group, processes have to agree on the current group membership. A membership service keeps all members updated on who the current members of the group are. Multicast view messages with group membership. View changes when processes join/leave the group. Each message is associated with a group view, the view the sender had when transmitting the message. Multicasts cannot pass across view changes, all multicast that are in transit while a view change occurs must be completed before the new view comes into effect. Implementing it: **1.** Views must be delivered in total order by each process belonging to each view. **2.** Normal operation within a view. Append the view number when receiving a multicast from a process belonging to the current view: If it matches the current view, the message is delivered, If it is higher, the message is queued until the matching view is installed, if it is lower, the message is discarded. **3.** Flush procedure triggered when a process P receives a new view message: Ask the application to stop multicasting messages until the new view is installed, forward all the messages delivered by P in the current view to all the members of that view, deliver any non-duplicated matching the current view received from any Q that belong to both views. After receiving ALL the forwarded messages from all the processes: Install the new view, Allow the application to resume multicasting messages. **Strict consistency:** Meet sequential specification of earch item x: Given 2 concurrent writes on x, one msut go first, a read returns value of the most recent, subsequent reads to the same value returns the same value or later. Operations run in a unique legal sequence that matches their real-time occurrence. Theoretical model impossible to achieve. (Writes must be instantly visible in all replicas and requires an absolute global time). **Linearizability:** Same goal as strict consistency but recognizing that operations take time to complete and can overlap. Operations appear to run in some common sequence that retains their real-time order when they do not overlap. Overlapping ops. Can take effect in any order, but their effect must become visible to everyone before they return (appearing to be atomic). Is compositional, separated linearizable histories can be composed in a single one. **Sequential consistency:** Operations appear to run in some common sequence that retains the program order for the operations of each process, real time is not taken into consideration. Non compositional. (See linearizability for more info). **Causal consistency:** Writes that are potentially causally related must be seen by all processes in the same order. **FIFO consistency:** Writes done by a single process are seen by all the others in the order in which they were issued. **Weak consistency:** Enforce consistency on groups of operations instead of individual reads/writes. Operate on synchronization variables to synchronize all the replicas. **Eventual consistency:** In absence of new updates, eventually all replicas will converge to identical values. Each client operates on a replica and updates propagate asynchronously. Writes might arrive in different orfers to all replicas: conflicts. Replcias must apply the updates in the same order. This requires consistent conflict resolution. **Last writer wins:** Order writes according to clock timestamp. Overwrite value only if timestamp of incoming write is higher than the current one. D: Some concurrent writes are silently dropped. D: Later writes can be overwritten due to clock skew. **Version vectors:** Each replica keeps a vector for each item with its version number per replica. Each incoming write includes the vector from a prior read. Replica compares both vectors to distinguish causally-related and concurrent writes. Dotted version vectors implement this idea. Each replica keeps a version vector and each incoming write includes the vector. **Read Your Writes:** If client C writes a data on replica A, any successive read on x by C on replica B will return the written value or a more recent one. Replica A updates the client's write set. Replica B checks WS before reading to ensure. **Monotonic Reads:** If client C reads a data item x on replica A, any successive read on x by C on replica B will return the same value or a more recent one. Replica A updates the client's read set. Replica B checks RS before reading to ensure. **Monotonic Writes:** A write by client C on a data item x on replica A is completed also in replica B before C performs any successive write on x on replica B. Replica A updates the client's write set. Replica B checks WS before writing to ensure. **Writes Follow Reads:** If C reads a data item x on replica A, relevant writes for that read are completed also in replica B before C performs any successive write on x on Replica B. Replica A updates the client's RS Replica B checks RS before writing. **Update propagation:** A. Propagate a notification to other replicas. Recipients invalidate their copy. B. Transfer the data from one replica to another. Recipients replace their copy. C. Propagate the update operation to other replicas. This is active replication. A. Push-based/Server-based approach: Updates are sent by the server. B. Pull-based/Client-based approach: Updates are requested by the client. **Adaptive leases:** A. Age-based leases: Lease depends on the last time the item was modified. An item that ahs not changed for a long time, will not change in a near future. B. Renewal-frequency-based leases: Lease depends on how often a specific client requests a cached item to be refreshed. C. State-based leases: Lease depends on state-space overhead at the server. **Primary-based protocols:** Each data item is associated with a primary replica which is in charge of coordinating write operations (If primary fails, one of the replcias is elected): **A.** Primary-backup remote-write: All writes are done at a fixed single replica, reads can be carried out locally. **B.** Primary-backup local-write protocols: Primary migrates to the replica that is writing successive writes are carried out locally, and then the replicas are updated using a non-blocking protocol. **Hierarchical process groups:** Use process replication to build a hierarchical process group that tolerates process failures. Implemented through a primary-based protocol. The coordinator acts as the primary. The workers act as backups. **1.** Clients send requests to the coordinator (reads might go to workers). **2.** Coordinator provides the requested service and updates the workers. **Replicated-write protocols:** Writes can be carried out at multiple replicas instead of only one, as occurs in primary-based protocols. **1.** Active replication: Clients broadcast each operation to all the replicas and they are carried out in the same order everywhere. Use total+fifo-ordered multicast. Senders waits until the multicast message is delivered back then it returns to the client. **2.** Quorum-based protocols: Clients musy get replies from a quorum of the N replicas before reading or writing. Nr+Nw>N to avoid read-write conflicts. Nw>N/2 to avoid write-write conflicts. To implement linerizability: Use version vectors. Perform read repair. Quorum Write **1.** Read phase: Send requests, await replies, learn the highest version. **2.** Write phase: Send requests, await ACK from Nw replicas, write is complete. Quorum Read: **1.** Read phase: Send requests, await repies, if all verion numbers are the same return. **2.** Write phase: (read repair) **Flat process groups:** Use process replication to build a flat process group that tolerates process failures. Voting. Implemented through a replicated-write protocol. Active replication: send requests to all workers and await one reply or K+1 identical replies. K+1 processes can tolerate K crash/omission failures. 2K+1 processes can tolerate K byzantine failures. Quorum-based: Clients send requests to all workers and await replies Nr of them. 2K+1 can tolerate K crash. 3K+1 can tolerate Byzantine.