1 Time

1.1 Physical Clocks

Synchronize at least every $R < \delta/2\rho$ to limit skew between two clocks to less than δ time units

1.1.1 Cristian Algorithm

Synchronize nodes with a server using UTC (Coordinated Universal Time) receiver within a specified bound, termed as External synchronization:

1. Each client queries the server for time at regular intervals R.
2. The client adjusts its time to $TS + \frac{\text{RTT}}{2}$.
   - Here, RTT denotes the round-trip time, calculated as $T2 - T1$.
   - assumes symmetrical latency.
   - The accuracy of the client's clock is $\pm(\frac{\text{RTT}}{2} - \text{Min})$.

1.1.2 Berkeley Algorithm

Maintain clock synchronization among entities within a bound (Internal synchronization):

1. Master polls slave clocks at intervals R.
   -Master adapts slave clocks considering round-trip times.
2. Master calculates a fault-tolerant average of slave clocks.
   -Discards clocks outside the bound.
3. Master transmits adjustments to local clocks.

1.2 Logical Clocks

1.2.1 Lamport

Each process P_i has a local counter C_i.

1. Event at P_i increments C_i.
2. P_i sets $\text{ts}(m) = C_i$ for message m.
3. Upon receiving m, P_j adjusts C_j and increments.

1.2.2 Vector Clocks

Each P_i has $\text{VC}_i[1 \dots N]$.

1. P_i increments $\text{VC}_i[i]$ and sends VC_i with m.
2. P_j updates VC_j on receiving m.

Vector clocks detect causality:

1. $\text{VC}(a) < \text{VC}(b)$ implies $a \to b$.
2. If neither $\text{VC}(a) < \text{VC}(b)$ nor $\text{VC}(b) < \text{VC}(a)$, $a \parallel b$.

2 Chandy-Lamport algorithm

Assumptions:

- Reliable processes and channels.
- Unidirectional, FIFO channels.
- Strongly connected topology.
- Processes execute during snapshot.
- Any process initiates a snapshot.

Steps by initiator:

- Record state.
- Send marker on every channel.
- Record messages until markers received.

On receiving marker over c:

1. If no state recorded:
   (a) Record state.
   (b) Send marker on every channel.
   (c) Record c as empty set.
   (d) Record other channels until markers.
2. Otherwise, record state of c as set of messages received over c since it saved its state.

Finish after recording state and all incoming channels.

3 Election algorithms

3.1 Bully Algorithm

Assumptions:

- The system is synchronous(It can use timeouts to detect process failures and processes not responding to requests)
- Topology is a strongly connected graph: there is a communication path between any two processes
- 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

Algorithm:

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 wins and sends a Coordinator message to the rest

3.2 Chang and Roberts

Assumptions:

- 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

3.3 Enhanced Ring

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 (i.e., the first process that detects its ID in the message), 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:
   - If the elected process is in the ID list, the election is over.
   - Everyone knows who the coordinator is and who the members of the new ring are.
   - Otherwise, the election is reinitiated.

4 Multicast Communication

4.1 Basic Reliable Multicast

1. Sender P assigns sequence number SP to each outgoing message.
2. P stores each outgoing message in a history buffer, removing it only after acknowledgment from all recipients.
3. Each Q tracks the sequence number $LQ(P)$ of the last message from P.

- Upon receiving a message from P, Q follows:
  - If $SP = LQ(P) + 1$:
    - Q delivers, increases $LQ(P)$, and acknowledges to P.
  - If $SP > LQ(P) + 1$:
    - Q queues the message, requests missing ones, and delivers upon alignment.
  - If $SP \le LQ(P)$:
    - Q discards as it has delivered the message.

4.2 Scalable Reliable Multicast

- Only missing messages are reported (NACK) - NACKs are multicast to all the group members - Successful delivery is never acknowledged
- 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

4.3 Ordered Multicast

remember to also deliver the message you are sending!

4.3.1 FIFO ordering

Using sequence numbers per sender:

1. A message delivery is postponed and stored in a hold-back queue until its sequence number corresponds to the next expected number.
2. For more detailed information, please refer to the 'basic reliable multicast'.

4.3.2 Total ordering

Using sequence numbers specific to each group:

1. Send messages to a sequencer, which then multicasts them with sequential numbering.
   (Message delivery is deferred and stored in a hold-back queue until its sequence number is reached)
2. Processes collaborate to determine sequence numbers (A_j is the largest agreed number that he has received so far):
   (a) The sender multicasts message m.
   (b) Each receiver j replies with a proposed sequence number for message m (including its process ID) that is $P_j = \text{Max}(A_j, P_j) + 1$ and places m in an ordered hold-back queue according to P_j.
   (c) The sender selects the largest of all proposals, N, as the agreed number for m and multicasts it.
   (d) Each receiver j updates $A_j = \text{Max}(A_j, N)$, tags message m with N, and reorders the hold-back queue if needed.
   (e) A message is delivered when it is at the front of the hold-back queue and its number is agreed.

4.3.3 Causal ordering

Utilizing Vector Clocks: A message is only considered delivered if all causally preceding messages have been delivered.

1. P_i increments $\text{VC}_i[i]$ only upon sending a message.
2. If P_j receives a message m from P_i, it delays the delivery until certain conditions are satisfied:
   (a) $\text{VC}(m)[i] = \text{VC}_j[i] + 1$ - This means m is the next expected message from P_i.
   (b) $\text{VC}(m)[k] \le \text{VC}_j[k]$ for all $k \ne i$ - Ensures that P_j has seen all messages seen by P_i before m.
3. After delivering m, P_j increments $\text{VC}_j[i]$.

5 Consensus

5.1 Dolev & Strong's Algorithm

The algorithm progresses over $f + 1$ rounds:

1. Each process initially proposes a value.
2. From round 1 to round $f + 1$, every process:
   (a) Multicasts new values (those not transmitted in prior rounds). Initially, it sends its proposed value.
   (b) Gathers values from other processes and logs any novel values received.

   A round concludes either when values from all processes are assembled or upon a timeout, determined by the maximum message latency.
3. Each process finalizes its decision based on the accumulated values.

5.2 OM Algorithm

OM(m): A solution with oral messages with at most m traitors and at least $3m+1$ generals. Assumptions:

1. Sent messages are delivered correctly (no corruption).
2. The absence of a message can be detected.
3. The receiver of a message knows who sent it.

OM algorithm is recursive: OM(0):

1. The commander sends his value to every lieutenant.
2. Each lieutenant accepts the value he receives as the order from the commander

OM(m), $m > 0$:

1. The commander sends his value to every lieutenant.
2. Each lieutenant acts as a commander in OM($m-1$) to broadcast the value he got from the commander to each of the remaining $n-2$ lieutenants.
3. Each lieutenant accepts the majority value $\text{majority}(v_1, v_2, \dots, v_{n-1})$ as the order from the commander.

Definitions:

- v_i: Value directly received from the commander in 'step 1'.
- $v_j, \forall j \in \{1, n-1\}, i \neq j$: Values indirectly received from the other lieutenants in 'step 2'.
- If a value is not received, it is substituted by a default value.

6 Concurrency Control

6.1 Strict Two-Phase Locking

1. When a read or write operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds (where promotion is prevented by a conflicting lock, rule b is used).
2. When a transaction commits or aborts, the server unlocks all objects it locked for the transaction.

6.2 Timestamps

- Each transaction has a unique timestamp, which:
  - Defines its position in the sequence of transactions.
  - Is assigned when the transaction starts.
- Each object x maintains the following information:
  1. Write timestamp for its committed value: $\mathrm{wts}(x)$.
  2. Read timestamps, represented by the highest: $\mathrm{rts}(x)$.
  3. Pending read and tentative write operations, along with their corresponding timestamps.
- Operations are validated upon execution by comparing object and transaction timestamps.
- Transaction T performs a write operation on x:
  - Valid only if object x was last read and written by earlier transactions, satisfying the conditions: $(\mathrm{ts}(T) \geq \mathrm{rts}(x))$ AND $(\mathrm{ts}(T) > \mathrm{wts}(x))$.
  - Add to the tentative writes a new version of the object with a write timestamp set to $\mathrm{ts}(T)$.
  - Overwrite the version if it already exists.
  - Tentative versions are maintained in order based on their timestamps.
  - If the write operation is not valid, transaction T is aborted because a transaction with a later timestamp has already read or written the object.
- Transaction T performs a read operation on x:
  - Valid only if object x was last written by an earlier transaction, satisfying: $\mathrm{ts}(T) > \mathrm{wts}(x)$.
  - Must read the version with the highest write timestamp lower than the transaction one.
  - If this is still tentative, the read stays pending until the earlier transaction completes.
  - Otherwise, read the committed version and set $\mathrm{rts}(x)$ to $\max\{\mathrm{rts}(x), \mathrm{ts}(T)\}$.
  - If T has already written the object, this will be used.
  - If the read operation is not valid, transaction T is aborted.
- Transaction T performs a commit operation:
  - For each object x written by T, replace the committed version with the tentative one and set $\mathrm{wts}(x)$ to $\mathrm{ts}(T)$.
  - If x has any pending read, write, or commit operation with a lower timestamp than T, the commit remains pending until those earlier operations complete.
  - Otherwise, the commit proceeds and also resumes any pending read and commit operations waiting for T.

6.3 Optimistic Concurrency Control

6.3.1 Working phase:

- Transaction keeps a tentative version of each object that it updates. This allows the transaction to abort with no effect on the objects.
- Transaction also keeps the read and write set:
  - Read set records the objects read by the transaction.
  - Write set records the objects written by the transaction.
- Read operations are directed to the tentative version if it exists; otherwise, they are directed to the committed one.
- Write operations are only performed onto the tentative version.

6.3.2 Validation phase:

- A transaction is given a sequence number when entering the validation phase, but real assignment is postponed until successful validation and update.
- On transaction T_{vcommit}, check for conflicts with overlapping T_i transactions (those not yet committed at the start of T_{v}).
- If the transaction is valid, it is allowed to commit; otherwise, abort the transaction or perform some conflict resolution

6.3.3 Update phase:

Backward validation:

- Validate a transaction T_v by comparing its read set with the write set of (committed) transactions T_i with sequence numbers $T_{\mathrm{start}} < T_i \leq T_{\mathrm{end}}$.
  - T_{start} is the highest sequence number assigned when transaction T_v enters the working phase.
  - T_{end} is the highest sequence number assigned when transaction T_v enters the validation phase.
- If a conflict is detected, abort the transaction T_v. All write sets of overlapping transactions must be maintained until their last concurrent transaction finishes.

Forward validation:
Validate a transaction T_v by comparing its write set with the read set of overlapping active (uncommitted) transactions.

1. Abort the transaction being validated.
2. Abort the conflicting active transactions.
3. Allow the conflicting transactions to complete and then try validation again.

Dynamic read sets must be consistently checked, e.g., block reads of active transactions on contended objects during the validation and update phases.

6.4 Distributed Commit

1. Coordinator sends a Vote-request to all participants, including itself.
2. Each participant replies with Vote-commit if it can commit locally; otherwise, it replies with Vote-abort and directly aborts.
3. Coordinator collects all the votes, including its own. If everyone voted to commit, it sends Global-commit; otherwise, it sends Global-abort.
4. Participants either COMMIT or ABORT according to the received message and optionally send an ACK when done.

6.5 Version Vectors

Each replica maintains a vector for each item, indicating its version number per replica.

- Increment its own version number with every write operation.
- Each incoming write includes the vector from a prior read, known as the causal context.
- The replica compares both vectors to differentiate between causally-related and concurrent writes.
  - Overwrite values from writes that happened before.
  - Retain values from concurrent writes as siblings.

Dotted version vectors implement this idea:

- Each replica k maintains a version vector VV_k for each item, represented as $\{(r_1, i_1, j_1), \ldots, (r_n, i_n, j_n)\}$.
  - Here, $(r_k, \text{range of events}\{1, i_k\}[, \text{last event} j_k])$.
  - For example: $\{(a,2),(b,1),(c,2,4)\} \equiv \{a_1, a_2, b_1, c_1, c_2, c_4\}$.
- When an incoming write occurs, it includes the vector VV_{inc}. Otherwise:
  - $VV_{\mathrm{new}} = \mathrm{merge}(VV_{\mathrm{inc}}, VV_k)$
  - Increment VV_{new}.
  - Obtain the last event of VV_{new} as a dot (using causal context from VV_{inc}).
  - Add the new value as a sibling.
  - Prune all siblings s for which $VV_{\mathrm{inc}} \geq VV_k(s)$.
- If $VV_{\mathrm{inc}} \geq VV_k$:
  - Increment VV_{inc}.
  - Obtain the last event of VV_{inc} as a dot.
  - Write the incoming value as the sole value (pairwise maximum).

7 quorum-based protocols

Clients must receive responses from a quorum of the N replicas before performing either a read (N_R)) or a write (N_W)).

- read-write: $N_R + N_W > N$
- write-write: $N_W > \frac{N}{2}$

7.1 Quorum Write

1. Read phase
   - Send read requests to all (or N_R) replicas.
   - Await replies from N_R replicas.
   - Learn the highest version among the replicas. (Merge siblings if needed)
2. Write phase
   - Send write requests (containing the version learned) to all (or N_W) replicas. If N_W, remaining replicas are updated as a background task (anti-entropy).
   - Await ACKs from N_W replicas.
   - Write is complete and can return to the user code.

7.2 Quorum Read

1. Read phase
   - Send read requests to all (or N_R) replicas.
   - Await replies from N_R replicas.
   - If all version numbers are the same, return.
2. Write phase ('read repair')
   - Otherwise, send write requests (containing the largest version number and the associated up-to-date value) to all (or only outdated) replicas.
   - Await ACKs from N_W (or only updated) replicas.
   - Return the value to the user code.

8 Flat Process Groups

Use process replication to build a flat process group that tolerates process failures:

- Processes are identical, and the group response is defined through voting.

8.1 Active Replication

- Clients send requests to all workers and await one reply (crash) or $K+1$ identical replies from workers(Byzantine).
- K-fault tolerance on worker failure:
  - $K+1$ processes can tolerate K crash/omission failures.
  - $2K+1$ processes can tolerate K Byzantine failures.

  (K failing processes could generate the same wrong reply; hence, $K+1$ correct processes are needed.)

8.2 Quorum-Based

- Clients send requests to all workers and await replies from N_R (or N_W) of them.
- K-fault tolerance on worker failure:
  - $2K+1$ and $2K$ processes can tolerate K crash or omission failures for writes and reads, respectively:
    - $N_W = \frac{N}{2} + 1$; $N = K + N_W \Rightarrow N > 2K$
    - $N_R + N_W = N + 1$; $N = K + N_R \Rightarrow N \geq 2K$
  - $3K+1$ and $3K$ processes can tolerate K Byzantine failures for writes and reads, respectively:
    - $N_W = \frac{N+K}{2} + 1$; $N = K + N_W \Rightarrow N > 3K$
    - $N_R + N_W = K + N + 1$; $N = K + N_R \Rightarrow N \geq 3K$