

**Diameter** Maximum distance between 2 nodes in network

**Radius** Half the diameter

**Spanning tree** A subgraph which is a tree and reaches all nodes. Has  $N - 1$  edges.

### Network Complexities

**Star network** Central server, everything else connected to it.  $O(n)$  messages.

**Chain network** Nodes connected to chain, need to go through  $n + i$  nodes to reach master.  $O(n^2)$  messages if every sends value,  $O(n)$  if aggregated.

**Tree network**  $O(|E|)$  where  $|E|$  is number of edges.

### Global Message Broadcast

- Flooding for Broadcast
  - Flood type, Unique ID, Data.
  - Send to all neighbours, if seen before discard otherwise forward and add to *seen list*.
  - Each node needs to store flood IDs.
  - Message complexity is  $O(|E|)$ , at worst  $O(n^2)$ .
  - To reduce complexity slightly, don't send to where you received from.
  - Time complexity is *diameter of G*.

### Computing Tree from a network

- BFS tree search
  - Every node has a parent pointer
  - Zero or more child pointers
  - Flood at every node, parent is the one who contacted it first.
  - If many, choose one with smallest id
  - Child informs parent of selection, parent creates child pointer.
  - Message complexity is  $O(|E|)$ , at worst  $O(n^2)$ .
  - Time complexity is *diameter of G*.

### Tree Based Broadcast

- Send message to all nodes using tree.
- Receive from parent, send to children.
- Message complexity is  $O(n - 1)$ .

### Aggregating Sum of Values Using BFS (Convergecast)

- Start from leaves.
- Every node waits for values from children, sum, and send up.
- Without the tree
  - Every node waits for  $O(|E|)$  messages.
  - $O(n|E|)$  messages in total.
  - Good fault tolerance.
- With the tree

- Any node can use broadcast.
- Bad fault tolerance: need to rebuild.
- Shortest path: from any node  $q$ , follow parent pointers to root.

### **BFS Trees for Routing**

- Create a BFS tree at every node.
- Store parent pointers to other nodes' BFS trees.
- $O(n|E|)$  message complexity for routing table.

### **Bit Complexity**

- Sometimes we calculate the amount of bits exchanged to assess complexity.
- Each node needs  $\Theta(\log n)$  bits to store ID

**Minimum Spanning Tree** Spanning tree with lowest sum of edge weights  $w(e)$ . Using it for broadcast has the smallest possible cost. Useful in point to point routing.

### **Cut Optimality**

- Every edge of the MST partitions the graph into 2 disjoint sets.
- No other edge can have smaller weight than the MST edge.
- Every non-MST edge when added to MST creates a cycle.

### **Prim's Algorithm**

- Initialise  $P = x$  and  $Q = E$ .
- While  $P \neq V$ 
  - Select edge  $(u, v)$  in the cut  $(P, V \setminus P)$  with smallest weight
  - Add  $v$  to  $P$
- If we search for minimum each time it's  $O(mn)$
- If we use heaps it's  $O(m \log n)$  or  $O(m + n \log n)$

### **Distributed Prim's Algorithm**

- In every round, find the minimum edge
- Use a convergecast every round for  $n$  rounds
- Complexity?
- Does not use distributed computation.
- Tree spreads from one point, rest of network is idle.

### **Kruskal's Algorithm**

- Each node is its own tree
- Sort all edges by weight.
- For each tree
  - Find the least weight boundary edge.
  - Add it to the set of edges: merges two trees into one
- To know which edge is boundary:

- Maintain ID for each tree.
- Check that endpoint has different tree ID.
- Update tree ID of all nodes when merging (smaller tree). The cost of updating IDs is  $O(n \log n)$ .

### **GHS Distributed Algorithm**

- In level 0 each node its own tree.
- Each tree has a leader (leader id = tree id).
- At each level  $k$ :
  - All leaders do a convergecast to find minimum boundary edge.
  - It then broadcasts this in the tree so the node knows.
  - The node informs the node on “the other side” which informs the leader.
- Possibly merging more than 2 trees at the same time.
- We get tree of trees: no cycles.
- Complexity
  - $O(n \log n)$  time
  - $O(n \log n + |E|)$  messages
- Weights need to be unique: use IDs to resolve ties

**Independent Set** A subset of vertices in the network such that no two vertices are connected by an edge of the network.

### **Maximum Independent Set**

- Largest such set, can be used for interference-free transmission in Wi-Fi.
- NP-hard to compute this set.

### **Maximal Independent Set**

- No more nodes can be added to it while keeping it an IS.
- Local:
  - Start with  $Q = \{v\}$
  - Repeat while  $Q$  non-empty:
    - \* Choose a node  $p$  in  $Q$
    - \* Put  $p$  in IS
    - \* Remove all neighbours from  $Q$
- Distributed:
  - Select root
  - Remove neighbours of root from possibility
  - Select IS in neighbours of neighbours etc.
- It could be pretty bad compared to the optimal Maximum IS.

---

**UTC** Universal Coordinated Time. Kept within 0.9s of Greenwich

**Piezoelectric effect** Squeeze a quartz crystal: generates electric field. Apply electric field: crystal bends.

**Quartz crystal clock** Resonates like a tuning fork. Accurate to parts per million

**Skew** Time difference between 2 clocks.

**Drift** Difference in rate between 2 clocks.

### Detecting a clock skew

- It is 5s behind
  - Advance by 5s to correct.
- It is 5s ahead
  - Pushing back is bad: could be received before sent.
- Monotonicity: time is always increasing
  - If behind, decrease clock rate.
  - If ahead, increase clock rate.

### How Clocks Synchronise

- Get time from server
- Delays in message transmission
- Delays due to processing time
- Server's time may be inaccurate

### Christian's Algorithm

- Request sent at  $T_0$ , reply received at  $T_1$
- Assume delays are symmetric,  $T_{server}$  is time from reply
- $T_{new} = T_{server} + (T_1 - T_0)/2$
- If minimum message transit time  $T_{min}$  is known
  - Range:  $T_1 - T_0 - 2T_{min}$ , accurate within  $(T_1 - T_0 - 2T_{min})/2$

### Berkeley Algorithm

- Assume no machine has perfect time
- Takes average of participants
- Sync everyone to average
- Master-slaves pattern
  - Master polls each machine for time
  - Computes average
  - Send each clock the offset to adjust time
- Fault tolerance
  - Ignore slaves with large skews
  - If master fails, elect new one

### Network Time Protocol

- Enable clients to synchronise to UTC
- Reliable: Redundant servers and paths
- Scalable: Enable many clients to sync frequently
- Security: Authenticate sources

- Servers in layers
    - Layer 1: Directly connected to atomic clock
    - Layer 2: Few  $\mu s$  off layer 1
  - Uses multiple rounds of messages, large number of servers and MST for inter-server sync
- 

## Logical Clocks

- Determine what happened before what without clocks.
- Use a counter at each process.
- Increment after each event.
- Can also increment when there are no events.
- Each event has an associated time

## Happened Before

- $a \rightarrow b$  means  $a$  before  $b$ .
- $a$  is send of message  $m$  and  $b$  is receive.
- Transitive property
- Events without a “happened before” relation are concurrent.
- Preserves causal relations.
- Implies partial order
  - Ordering between pairs of events.
  - No ordering between concurrent events.

## Lamport Clocks

- A logical clock
- Sent with every message
- On receiving a message, set own clock to  $\max(own, message) + 1$
- For any event  $e$ , write  $c(e)$  for logical time
- If  $a \rightarrow b$  then  $c(a) < c(b)$
- If  $a \rightarrow b$  then no Lamport clock exists with  $c(a) == c(b)$
- If  $e_1 || e_2$  then there exists a Lamport clock such that  $c(a) == c(b)$
- If we order all events by their Lamport clock then we get partial order satisfying causal relations
- Total order from Lamport clocks
  - If event  $e$  occurs in process  $j$  at time  $c(e)$
  - Give it time  $(c(e), j)$
  - Order events by  $(c, id)$

## Vector Clocks

- If  $a \rightarrow b$  then  $c(a) < c(b)$ .
- Also if  $c(a) < c(b)$  then  $a \rightarrow b$ .
- Each process  $i$  maintains a vector  $V_i$ .
- $V_i$  has  $n$  elements
  - keeps clock  $V_i[j]$  for every other process  $j$
  - On every local event:  $V_i[i] = V_i[i] + 1$

- On sending a message at  $i$ 
  - \* Adds 1 to  $V_i[i]$
  - \* Sends entire  $V_i$
- On receiving a message at  $j$ 
  - \* Take max element by element
  - \*  $V_j[k] = \max(V_j[k], V_i[k])$  for all  $k$
  - \* Adds 1 to  $V_j[j]$  (local event)
- $V == V'$  iff  $V[i] == V'[i]$  for all  $i$
- $V < V'$  iff  $V[i] < V'[i]$  for all  $i$
- $V \leq V'$  iff  $V[i] \leq V'[i]$  for all  $i$
- $a \rightarrow b$  if  $V(a) < V(b)$
- Two events are concurrent if neither  $<$  nor  $>$  is true
- Drawbacks
  - Entire vector sent with message
  - All vector elements ( $n$ ) have to be checked
  - $\Omega(n)$  per message communication complexity, increases with time

### Distributed Snapshots

- Take a snapshot of the system
- Global state: state of all processes and comm. channels
- Consistent cuts: set of states of all processes is a consistent cut if: for any states  $s, t$  in the cut  $s || t$ .
- If  $a \rightarrow b$ , then  $b$  cannot be before cut and  $a$  after cut

### Distributed Snapshot Algorithm

- Ask each process to record state.
- The set of states must be a consistent cut.
- Assumptions
  - Communication channels are FIFO
  - Processes communicate only with neighbours
  - We assume for now that everyone is a neighbour
  - Processes do not fail

### Chandy and Lamport Algorithm

- Send Rule at  $i$ 
  - Process  $i$  records state
  - On every outgoing channel where a marker has not been sent  $i$  sends a marker on the channel before sending any other message.
- Receive Rule at  $i$  on channel  $C$ 
  - $i$  has not received a marker before
    - \* Record state of  $i$
    - \* Record state of  $C$  as empty
    - \* Follow Send Rule
  - Otherwise

- \* Record state of  $C$  as set of messages received on  $C$  since recording  $i$ 's state and before receiving marker on  $C$ .
- Algorithm stops when all processes have received marker on all channels.
- $O(l)$  message complexity:  $l$  is number of links, plus the messages sent by normal execution of processes
- $O(d)$  time complexity:  $d$  is diameter

### Snapshot Properties

- If  $s_1$  (in  $p_1$ )  $\rightarrow$   $s_2$  (in  $p_2$ )
  - Then  $s_2$  before cut  $\implies$   $s_1$  before cut
  - Proof by contradiction:  $s_1$  after cut
    - \*  $p_1$  recorded its state before  $s_1$
    - \* Message  $m$  from  $p_1$  to  $p_2$ : this causes  $s_1 \rightarrow s_2$  to be true
    - \*  $p_1$  must have recorded state before sending  $m$
    - \*  $p_1$  must have sent marker to  $p_2$  before sending  $m$
    - \*  $p_2$  must have received marker before  $m$  and before  $s_2$
    - \*  $s_2$  must be after cut: contradiction

### Application of Snapshots

- Detection of stable predicates
- A property that once it becomes true, stays true
- Examples
  - Deadlocked: every process in some subset is waiting for another
  - Terminated: once ended, computation remains stopped
  - Loss of token: in mutex, process with token can access a resource. If token gets lost, it stays lost.
  - Garbage: If no-one has a reference to a file, that file can be deleted
- So if such a property was true before the snapshot, it is true in snapshot, and can be detected by checking the snapshot

### Non-stable Predicates

- Predicate may have happened but state has changed, e.g. "Was this file opened at some time?"
- Two types
  - Possibly  $B$ :  $B$  could have happened
  - Definitely  $B$ :  $B$  definitely happened
- Collecting global states
  - Each process notes its every state & vector timestamp
  - Sends it to server
  - We only need to save state changes affecting the predicate
  - The server looks at these and tries to figure out if predicate  $B$  was possibly or definitely true

### Possible States

- Server checks for possible states: consistent cuts for  $B$
- Create a lattice where a downward path from initial state to final state is a valid execution
- **Possibly**  $B$  occurs on at least downward path
  - Do BFS search from start

- If there is one state with  $B$  true then possibly  $b$  is true
- **Definitely**  $B$  occurs on all downward paths
  - Do BFS search from start
  - Do not visit nodes with  $B$  true
  - If BFS reaches final state and  $B$  is not true there then definitely  $B$  is false, otherwise it is true
- Complexity for both is  $O(k^N)$  where  $k$  is max number of events at a single process

**Mutual Exclusion** Multiple processes should not use same resource at once. Restricts access to critical section to at most one process at a time.

**Critical Section** Part of code that uses the restricted resource

### Mutex Properties

- **Safety:** Two processes should not use critical section simultaneously.
- **Liveness:** Every live request for CS is eventually granted.
- **Fairness:** Requests must be granted in the order they are made.

### Mutex Algorithms Assumptions

- Only one resource
- All channels are FIFO

### Central Server Algorithm

- There is a coordinator that holds a *token* for the resource.
- Other processes send token requests to coordinator.
- Server puts incoming requests into a queue.
- Sends token to first process in queue.
- Process returns token when done.
- Advantages: simple and constant complexity per message
- Disadvantages:
  - Central point of failure
  - Central bottleneck
  - Does not preserve order in async systems
  - Coordinator must be elected

### Token Ring Algorithm

- Processes arranged in a ring
- Token is continuously passed in one direction
- If process does not need to enter CS, it passes token
- Otherwise it holds token, executes CS and then passes
- Disadvantages
  - Not in order
  - Long delay in getting token
  - One failure breaks ring
  - Passes token even without requests

### Lamport's Mutex Algorithm



- Every node  $i$  has a queue of requests (sorted by timestamps)
- Process  $i$  sends CS request
  - $REQUEST(timestamp, i)$  to all processes
  - Enters  $(timestamp, i)$  in own queue
- Process  $j$  receives  $REQUEST(timestamp, i)$ 
  - Send timestamped  $REPLY$  to  $i$
  - Enters  $(timestamp, i)$  in queue
- Process  $i$  enters CS if
  - $(timestamp, i)$  at head of own queue
  - Received  $REPLY$  from all processes
- To release CS: send  $RELEASE$  to all
- On receiving  $RELEASE$  at  $j$  remove  $(timestamp, i)$  from queue
- Complexity:  $3(n - 1)$  messages per CS request

### Ricart and Agrawala's Algorithm

- Node  $j$  does not send  $REPLY$  if  $j$  has request with timestamp lower than  $i$  request
- Node  $j$  delays the  $REPLY$  until after own  $RELEASE$
- Process  $i$  sends CS request
  - $REQUEST(timestamp, i)$  to all processes
- Process  $j$  receives  $REQUEST(timestamp, i)$ 
  - If  $j$  has no own outstanding requests earlier than  $timestamp$  or is not executing CS
    - \* Send  $REPLY$  to  $i$
    - \* Enters  $(timestamp, i)$  to own queue
  - Else keep  $(timestamp, i)$  pending
- Process  $i$  enters CS if it has received  $REPLY$  from all.
- To release CS: send  $RELEASE$  to pending processes.
- Complexity:  $2(n - 1)$  messages per CS request.

### Maekawa's Quorum Algorithm

- Instead of getting permission from all, get it from subset
- For each process  $i$  we have a voting quorum  $V_i$ 
  - For all  $i, j : V_i \cap V_j \neq \emptyset$
  - For all  $i : i \in V_i$
  - Voting sets are same size
  - Each node part of same number of sets
- Arrange nodes in a square grid
- Quorum for node  $i$  are all nodes in same row or column
- Any two quorums intersect
- Complexity:  $O(\sqrt{n})$  per CS request.

---

**Packets** Messages sent in (fixed-size) packets.

## **Local Area Networks**

- Medium: Broadcast
- Message from one computer to all other computers
- Ethernet LAN is a broadcast medium and so is Wireless LAN
- Advantages
  - Sending a common message to everyone is easy
  - Finding destination is easy: destination field
- Disadvantage: Medium access - multiple transmissions at the same time

## **Medium Access**

- Only one transmission at a time
- Protocols
  - TDMA: every node has a periodic slot
  - CSMA: see if anyone else transmitting, defer
  - ACKs are used to confirm transmission
- More complicated for wireless (hidden terminal)

## **Routing**

- Finding a path in the network
- Every node has a routing table
- Equivalent to a BFS tree at every node
- Smaller routing tables by combining addresses

## **Location-based Routing**

- Uses nodes' locations to discover paths
- Greedy algorithm: forward to neighbour closest

## **Transport Management**

- UDP: send packet, hope it gets delivered
  - not FIFO
  - used in streaming
- TCP: send packet, ensure arrival
  - is FIFO
  - waits for ACKs, otherwise resends
  - slows down packet stream when packets not ACKed (assumes routers dropping them)

## **Overlay Network**

- parts of the network sometimes ignored
- nodes that carry but not participate or edges not used
- used in P2P networks where communication only with known nodes

## **Computation**

- Synchronous

- Operation in rounds
- In a round, a node performs computation, and then sends messages
- All messages sent at the end of round  $x$  are delivered at start of round  $x + 1$
- Can be implemented with  $m + c$  duration
  - \* if message transmission time bounded by  $m$
  - \* Computation times are bounded by  $c$
  - \* Clocks are synchronised
- Easier to design, starting point for design
- Asynchronous
  - No synchronisation or rounds
  - Nodes compute and send at different speeds
  - No assumptions about speeds
  - Simplifying assumptions can be made e.g.
    - \* Channels are FIFO
    - \* Code blocks are atomic (uninterrupted by messages)
    - \* Either communication or computation bounded

## Failures

- Hardware failures
- Out of power failure
- Software failure
- Can be permanent/temporary
- Nodes can fail individually or together (correlated)

**Stopping Failure** Node stops working, assumptions about what it finished and who knows about failure.

**Byzantine Failure** Node behaves arbitrarily or as adversary.

**Link Failure** Can be noise (waves at similar frequencies) or interference (nodes nearby communicating).  
Channels can become silent & unusable, active & unusable, or active & erroneous message

## Security

- Unauthorised access/modification
- Attack on nodes: causing nodes to fail, reading data, or taking control
- Attack on links: blocking communication, reading channel data, corrupting data

## Mobility

- Movement makes it harder to design distributed systems
- Communication is difficult: delays, lost messages, edge weights change

---

## Leader Election

- Agreement is simpler with a master but single point of failure (SPOF)
- When one master fails, another takes over

## Failure Detectors

- Detecting crashed processes
- Detecting “working” is easier (they respond), detecting “failed” is harder.
- Unreliable detectors: reply with “suspected (failed)” or “unsuspected”.
- **Example**
  - Suppose all messages delivered within  $D$  seconds.
  - Then we can require heartbeat every  $T$  seconds to failure detector.
  - If a failure detector does not get heartbeat in  $T + D$  seconds, it marks process as “suspected” or “failed”.
- Synchronous: simple, send a message and wait for  $2D + \epsilon$  (no need for detector)
- If  $T$  or  $D$  too large: long failure timeouts
- If  $T$  too small: too much pressure on clients
- If  $D$  too small: “working” could get marked as “failed”

### Real World

- Both synchronous and asynchronous too rigid
- Have 2 values:  $D1$  and  $D2$
- Use probabilities: delivery time is a distribution, estimate probability of failure
- $a$ : probability process fails within time  $T$
- $b$ : probability a message not received in  $T + D$
- Want to estimate  $P(a|b)$  using Bayes Theorem

### Distinguished Leader

: Leader must have a property other nodes do not have: node with highest ID.

### Aggregation Tree Leader Election

- Node  $r$  detects leader failed, initiates election
- Node  $r$  creates BFS tree
- Asks for max node ID via aggregation (convergecast)
- If all  $n$  nodes start election needs  $n$  trees
  - $O(n^2)$  communication and  $O(n)$  memory per node

### Ring Leader Election (Chang and Roberts)

- Nodes send to right max of received from left and own ID
- When max ID node receives the ID it knows everyone has seen it and declares itself the leader
- If multiple elections at the same time: it sends the ID only if greater than own ID
- Message complexity:  $O(n^2)$

### Ring Leader Election (Hirschberg and Sinclair)

- $k - neighborhood$  of node  $p$ : set of all nodes within distance  $k$ .
- Message has a time-to-live (ttl) variable decremented on receiving. When zero, not forwarded.
- Algorithm operates in phases, each phase ttl is doubled
- Node sends messages right and left with ID and ttl
- Node returns message if ID in message greater and ttl is zero
- Otherwise it forwards it and decreases ttl
- If both returned, node is leader of  $k - neighborhood$
- When  $2^i \geq n/2$  only 1 process survives: the leader
- Number of rounds:  $O(\log n)$

- Number of messages:  $O(n)$  per phase

### **Bully Algorithm**

- Each node knows IDs of all nodes
- Synchronous (round) operation
- Node  $p$  initiates election
- $p$  sends message to all nodes with higher ID
- If  $p$  does not get any replies, it declares itself a leader
- Higher ID gets message, responds and starts election
- If higher ID gets leader declaration it starts election again
- Message complexity:  $O(n^2)$

### **Multicast**

- Send message to multiple nodes with only 1 message
- Happens only within a group (LAN)
- Nodes can accept message and join group (tree at each node)
- IP addresses from 224.0.0.0 to 239.255.255.255
- A message to one of the addresses sent to all nodes subscribed to the group (same network)
- When joining node informs OS which informs network: Internet Group Mgmt protocol (IGMP)
- IP Multicast
  - Sender sends only once
  - Every router forwards only once
  - Uses UDP: no guarantees

---

### **Reliable Multicast**

- Sending process is in multicast group
- Nodes may fail
- One to one communication between processes
- No network fails, no messages undelivered
- $multicast(g, m)$ : message  $m$  to group  $g$
- $receive(m)$ : OS receives message and gives to multicasting process
- $deliver(m)$ : multicast process delivers to application
- Integrity: A working process  $p$  in group  $g$  delivers  $m$  at most once, and  $m$  was multicast by some working process.
- Agreement: If a working process delivers  $m$  then all other working processes in group  $g$  will deliver  $m$

### **Basic Reliable Multicast**

- $send(m, q)$  to each process  $q$  in group
- On receive, pretend it was multicast
- Does not implement Agreement: sender could crash mid-send

### **Reliable Multicast Implementation**

- Initialize  $Received =$
- Send message to each process in group

- On receive if not in *Received*
  - Add it to *Received*
  - Forward  $m$  along
  - Deliver to application
- Satisfies Integrity:  $send(m, q)$  is reliable
- Satisfies Agreement: forwards before delivers

### **Multicast Ordering**

- We want messages delivered in correct order
- FIFO: if a process performs 2 multicasts, every process sees them in correct order
- Causal: if  $multicast(m) \rightarrow multicast(m')$  then deliver  $m$  before  $m'$  (implies FIFO)
- Total: All working processes deliver messages in same order
- Our Multicast is FIFO assuming it sends to group members in same order and channels are FIFO

### **Causally-ordered Multicast**

- Each process has a Vector clock
- Suppose  $p$  sends multicast  $m$
- $q$  receives  $m$  and holds until
  - It has delivered any earlier message by  $p$
  - It has delivered any multicast message delivered by  $p$  before  $m$
- Checking using vector timestamps

### **Totally-ordered Multicast**

- Using a sequencer process
  - Process  $p$  wants to multicast
  - It requests sequence number from sequencer
  - Send multicast with sequence
  - Multicasts are delivered by sequence number
  - SPOF and bottleneck
- Using collective agreement
  - Process  $p$  sends basic multicast to the group
  - Each process picks a sequence number
  - Processes run protocol to agree on sequence number
  - Messages delivered according to the agreement

### **Basic Consensus**

- Set of processes each with state undecided
- Termination: Set their decision variable and enter decided state
- Agreement: If processes entered decided state their decisions are equal
- Integrity: If all processes proposed value  $v$  then all of them have decision  $v$
- Simple algorithm
  - Use reliable multicast to communicate all values
  - Rule e.g. min or max decides

### Byzantine Generals Consensus

- Commander issues attack
- One or more processes may be faulty
- Termination: everyone decides
- Agreement: non-faulty processes agree
- Integrity: Non-faulty commander decides
- No solution with  $n < 3f$  processes

### Interactive Consensus

- Processes have to agree on a vector of values
- Each process contributes only to part of the vector but all processes have same vector in the end
- Termination: everyone decides
- Agreement: same vector  $V$
- If  $p_i$  proposes  $x$  then  $V_i = x$  for all

### Termination Detection

- Computation started by  $s$  by sending messages
  - Process  $s$  starts with weight 1.0
  - When any process sends a message it puts part of its weight in the message
  - When any process receives a message it adds weight to own weight
  - When a process finishes computing it sends current weight to  $s$
  - When  $s$  has weight = 1.0 it knows the deed is done
  - No message is lost (TCP required)
-