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 $(P, V\P)$ 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 μs 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

- $a \to b$ 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 $a \to b$ then $c(a) < c(b)$
- If $a \to b$ 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 $a \to b$ then $c(a) < c(b)$.
- Also if $c(a) < c(b)$ then $a \to b$.
- 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
- $V \le V'$ iff $V[i] \le V'[i]$ for all i
- $a \to b$ 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