Distributed Systems

Lecture 22

Monsoon 2024

International Institute of Information Technology

Hyderabad, India

This Lecture

- Study distributed algorithms, mostly for graph based computations.
- Start with system model, complexity measures, and then move to algorithms.
- End with some terminology and discussion.

System Model

- We model the distributed system as a network or a graph G.
- The processors in the distributed system act as nodes of the graph G.
- The communication links between the processors act as the edges of the graph G.
    - Bidirectional links correspond to undirected graphs
    - Directional links correspond to directed graphs.
- Various topologies possible, but we leave it as *any* topology.

Nature of Execution

- Centralized Vs. distributed algorithms
  - Centralized: asymmetric roles; client-server configuration; processing and bandwidth bottleneck; point of failure
  - Distributed: more balanced roles of nodes, difficult to design perfectly distributed algorithms (e.g., snapshot algorithms, tree-based algorithms)
- Synchronous Vs. Asynchronous algorithms

Nature of Execution

- Synchronous Vs. Asynchronous algorithms
  - Synchronous:
    - upper bound on message delay
    - known bounded drift rate of clock wrt. real time
    - known upper bound for process to execute a logical step
  - Asynchronous: above criteria not satisfied
    - spectrum of models in which some combination of criteria satisfied
- Algorithm to solve a problem depends greatly on these assumptions.

System Model

- Distributed systems inherently asynchronous
- Algorithms must ideally be designed in the asynchronous model.
- However, most algorithm designers use the synchronous model.
- Take advantage of constructs called synchronizers that can convert synchronous algorithms to asynchronous versions.

Complexity Measures

- In a sequential algorithm too, several metrics of complexity are available.
  - Time
  - Space
  - Disk Accesses
  - ...
- Similarly, also in distributed systems, several notions of complexity exist.
- These notions may also change slightly based on the model of the system.

Complexity Measures

- In distributed systems, several notions of complexity exist.
- These notions may change slightly based on the model of the system.
- Typical measures of complexity include:
  - Space complexity per node
  - System-wide space complexity
  - Time complexity per node
  - System-wide time complexity. Do nodes execute fully concurrently?
  - Message complexity
    - Number of messages (affects space complexity of message overhead)
    - Size of messages (affects space complexity of message overhead + time component via increased transmission time)
  - Message time complexity: depends on number of messages, size of messages, concurrency in sending and receiving messages

Complexity Measures

- Other metrics: # send and # receive events; # multicasts, and how implemented?
- (Shared memory systems): size of shared memory; # synchronization operations

First Algorithm

- Let us design a simple BFS algorithm in the synchronous setting.
- We assume that we are given a graph G with each node knowing who its neighbors are.
- We also assume that there is an initiator node, the source node of the BFS.
  - This node is known apriori.

Synchronous BFS

- The idea of the algorithm is as follows.
- BFS required nodes to be arranged into levels 0, 1, 2, ... such that a node in level i has a parent at level i – 1.
- The initiator marks itself to be at level 0.
- The initiator then sends a QUERY message to all its neighbors.
- The message will include the level number of the initiator.
- The neighbors will mark themselves as nodes at Level 1 and propagate their QUERY messages.

Synchronous BFS

- The initiator marks itself to be at level 0.
- The initiator then sends a QUERY message to all its neighbors. The message will include the level number of the initiator.
- The neighbors will mark themselves as nodes at Level 1 and propagate their QUERY messages.
- Some node may now receive two QUERY messages.

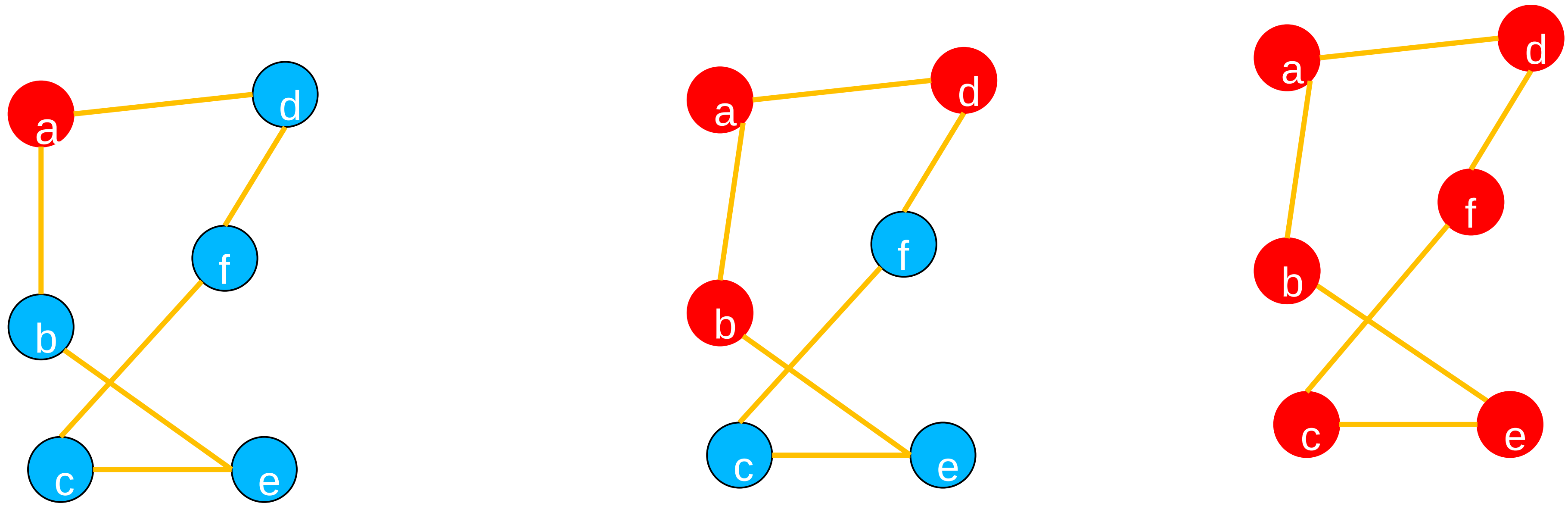
Synchronous BFS

- The initiator then sends a QUERY message to all its neighbors. The message will include the level number of the initiator as 0.
- The neighbors will mark themselves as nodes at Level 1 and propagate their QUERY messages.
- Some node may now receive two or more QUERY messages.
- Such a node can pick *any* of the message sources as its parent and set its level number correctly.

Synchronous BFS

- This process repeats until all nodes are marked with a valid level number.
- Each node can terminate when it gets a valid level number and sends out the QUERY message(s).

Synchronous BFS – Example



- Nodes in red color are marked as visited and nodes in blue color are unmarked. Node a is the source and hence nodes b and d are marked in the next round.
- Subsequently, the other nodes are marked in the third round as shown.

Synchronous BFS – Complexity

- Local space complexity : O(degree)
- Global space complexity = Sum of local space
- Local time complexity : O(diameter + degree)
- Number of messages : at most two per edge
- Size of each message : O(node id)

- Currently, we have suggested how a node can find its parent.
- Can modify the algorithm so that a node will find all its children as well.

Synchronous BFS

Algorithm 18 Synchronous BFS Algorithm

```
procedure BFS
    marked ← false, level ← 0, parent ← φ
    if SOURCE (u) then
        marked ← true, level ← 0
        Send ⟨mark, u, 1⟩ to all the neighbors of u
    end if

    for round ∈ [1 . . . D] do
        parallel
            repeat
                ▷ Node u receives a message from node v
                Receive ⟨mark, v, l_v⟩
                if marked = false then
                    parent ← v
                    marked ← true
                    level ← l_v + 1
                    Send ⟨mark, u, level⟩ to neighbors
                end if
            until Node u receives messages in the current round
        end parallel
    end for
end procedure
```

Asynchronous Spanning Tree

- Big difficulty to decide what are the children of a node.
- This difficulty affects deciding whether a node can terminate.
- Solved by using extra messages beyond QUERY.
  - ACCEPT and REJECT.
- Nodes flood a QUERY message once they find a parent and a valid level number.
- Once a node without a valid level number receives a QUERY messages, it accepts the sender of the first of such messages received as the parent.
  - Sends ACCEPT to parent,
  - Sends REJECT to all future QUERY messages received.

Asynchronous Spanning Tree

- Nodes flood a QUERY message once they find a parent and a valid level number.
- Once a node without a valid level number receives a QUERY messages, it accepts the sender of the first of such messages received as the parent.
  - Sends ACCEPT to parent,
  - Sends REJECT to all future QUERY messages received.
- A node can terminate only when it receives replies from all the neighbors that it sent a QUERY message to.

Asynchronous Spanning Tree

(local variables)
int $parent \longleftarrow \perp$
set of int $Children$, $Unrelated \longleftarrow \emptyset$
set of int $Neighbors \longleftarrow$ set of neighbors
(message types)
QUERY, ACCEPT, REJECT

(1) When the predesignated root node wants to initiate the algorithm:
(1a) **if** ($i = root$ **and** $parent = \perp$) **then**
(1b) **send** QUERY to all neighbors;
(1c) $parent \longleftarrow i$.

(2) When QUERY arrives from j:
(2a) **if** $parent = \perp$ **then**
(2b) $parent \longleftarrow j$;
(2c) **send** ACCEPT to j;
(2d) **send** QUERY to all neighbors except j;
(2e) **if** $(Children \cup Unrelated) = (Neighbors \setminus \{parent\})$ **then**
(2f) **terminate**.
(2g) **else send** REJECT to j.

Asynchronous Spanning Tree Program

(3) When ACCEPT arrives from j:
(3a) $\mathit{Children} \longleftarrow \mathit{Children} \cup \{j\}$;
(3b) **if** $(\mathit{Children} \cup \mathit{Unrelated}) = (\mathit{Neighbors} \setminus \{\mathit{parent}\})$ **then**
(3c) **terminate**.

(4) When REJECT arrives from j:
(4a) $\mathit{Unrelated} \longleftarrow \mathit{Unrelated} \cup \{j\}$;
(4b) **if** $(\mathit{Children} \cup \mathit{Unrelated}) = (\mathit{Neighbors} \setminus \{\mathit{parent}\})$ **then**
(4c) **terminate**.

Asynchronous Spanning Tree

- Complexity
  - Local space complexity: Degree of the node
  - Global space: O(Σ local space)
  - Local time: O(degree)
  - Message complexity: ≥ 2, ≤ 4 messages/edge. Thus, [2l, 4l], l is the number of links in the graph.
  - Message time complexity: d + 1 message hops. (d is the diameter)
  - Spanning tree: no claim can be made. Worst case height n − 1

Asynchronous Spanning Tree

- Example – To do

Bellman-Ford Algorithm

- The famous Bellman-Ford algorithm for single-source-shortest-paths can be run in a distributed setting in a near straight-forward manner.
- Input is a weighted graph, no cycles with negative weight
- No node has global view; only local topology; synchronous
- Assumption: node knows n; needed for termination
- After k rounds: length at any node has length of shortest path having k hops
- After k rounds: length of all nodes up to k hops away in final shortest path tree has stabilized.

Bellman-Ford Algorithm

Pseudo-code

(local variables)
int $length \longleftarrow \infty$
int $parent \longleftarrow \perp$
set of int $Neighbors \longleftarrow$ set of neighbors
set of int $\{weight_{i,j}, weight_{j,i} \mid j \in Neighbors\} \longleftarrow$ the known values of the weights of incident links

(message types)
UPDATE

(1) **if** $i = i_0$ **then** $length \longleftarrow 0$;
(2) **for** $round = 1$ **to** $n - 1$ **do**
(3) **send** UPDATE(i, $length$) to all neighbors;
(4) **await** UPDATE(j, $length_j$) from each $j \in Neighbors$;
(5) **for** each $j \in Neighbors$ **do**
(6) **if** $(length > (length_j + weight_{j,i})$ **then**
(7) $length \longleftarrow length_j + weight_{j,i}$; $parent \longleftarrow j$.

Bellman-Ford Algorithm

- Termination: $n - 1$ rounds
- Time Complexity: $n - 1$ rounds
- Message complexity: $(n - 1) \cdot l$ messages

More Algorithms

- Some algorithmic steps are known to be powerful primitives, a. la. subroutines, that can be useful in distributed algorithm design.
- Two such are: Broadcast and Convergecast.

More Algorithms

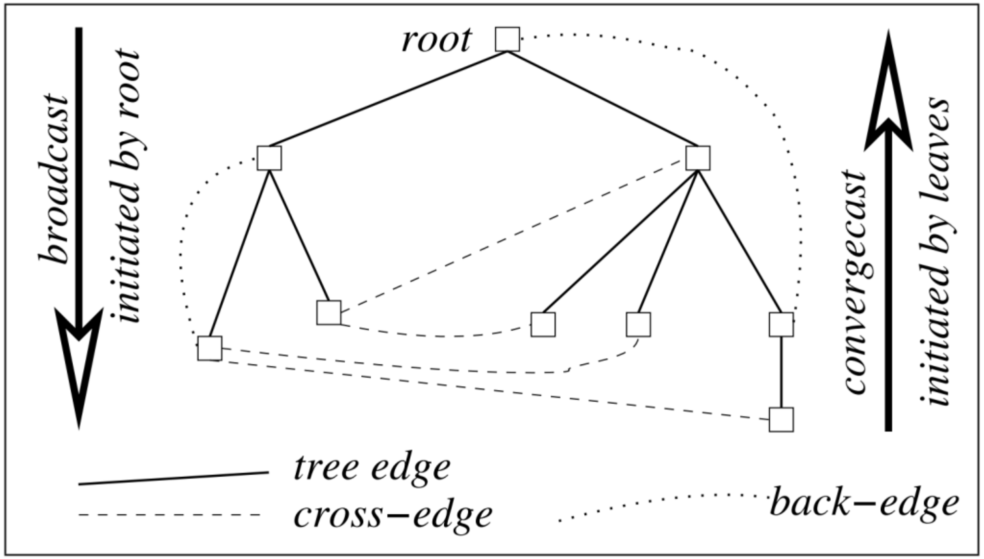


Figure 5.5: Tree structure for broadcast and convergecast

- Some algorithmic steps are known to be powerful primitives, a. la. subroutines, that can be useful in distributed algorithm design.
- Two such are: Broadcast and Convergecast.
- Broadcast: distribute information
  - BC1: Root sends info to be broadcast to all its children. Terminate.
  - BC2: When a (nonroot) node receives info from its parent, it copies it and forwards it to its children. Terminate.

More Algorithms

- Some algorithmic steps are known to be powerful primitives, a. la. subroutines, that can be useful in distributed algorithm design.
- Two such are: Broadcast and Convergecast.
- Convergecast: collect info at root, to compute a global function
  - CVC1. Leaf node sends its report to its parent. Terminate.
  - CVC2. At a non-leaf node that is not the root: When a report is received from all the child nodes, the collective report is sent to the parent. Terminate.
  - CVC3. At root: When a report is received from all the child nodes, the global function is evaluated using the reports. Terminate.

More Algorithms

- Uses: compute min, max, leader election, compute global state functions
- Time complexity: $O(h)$; Message complexity: $n - 1$ messages for BC or CC
- In the above, h is the height of the tree used.

Minimum Spanning Tree

- Given a weighted connected graph on n nodes, find a spanning tree of minimum total weight.
  - The tree that minimizes the sum of the weights of the edges in the spanning tree.
- Traditional algorithms include
  - Kruskal’s algorithm
  - Prim’s algorithm
- These do not work well in the distributed setting.
- A popular algorithm is that of Gallager-Humblet-Spira, which we study now in brief.

Minimum Spanning Tree

- A popular algorithm is that of Gallager-Humblet-Spira, which we study now in brief.
- Let T be the MST of a graph G = (V, E, W)
- Some common notions in most distributed MST algorithms are:
  - An MST fragment F of T is a connected subgraph of T.
  - An outgoing edge of an MST fragment F is an edge e in E such that one end point of the e is in F and the other is not.
  - The minimum-weight outgoing edge (MOE) of a fragment F is the edge with minimum weight among all outgoing edges of F.

Minimum Spanning Tree

- Check for yourself: The MOE of a fragment F = (V_F, E_F) is an edge of the MST T.
- The GHS algorithm operates in phases.
- In the first phase, the algorithm starts with each individual node as a fragment by itself.
- At the end, there is only one one fragment which is the MST.
- All fragments find their MOE's simultaneously in parallel.

Minimum Spanning Tree

- Invariant across Phases: Each MST fragment has a leader and all nodes know their respective parents and children.
- The root of the tree will be the leader.
- Each fragment is identified by the identifier of its root called the fragment ID.
- Each node in the fragment knows its fragment ID

Minimum Spanning Tree

- Here is how one Phase of the algorithm operates.
- Two operations in each phase
  - Find MOE of all fragments, and
  - Merging fragments via their MOEs.

Finding MOE

- The root of the fragment broadcasts a message <Find MOE> to all nodes in the fragment using the edges in the fragment.
- When a node receives a <Find MOE> message, it finds its minimum outgoing incident edge, i.e. the minimum weight outgoing edge among all the incident edges.
  - This can be done by checking the neighbors of the node.
- Then, each node sends its minimum outgoing incident edge to the root by using the ConvergeCast routine.
- The root then finds the MOE as the minimum among all the edges convergecast.

Merging Fragments

- In this phase, fragments are merged via their MOEs.
- Once the leader finds the MOE, it broadcasts a <Merge MOE> message to all its fragment nodes
- When a node receives the Merge message, it knows whether it is has the MOE edge incident on it.
- If so, it sends a “Request to combine" message to its neighbor which is the other end point of the MOE edge.

Minimum Spanning Tree

- The node with the higher identifier becomes the root of the combined fragment.
- The (combined) root broadcasts a <NEW-FRAGMENT> message through the fragment edges and the MOE edges chosen by all the fragments.
- Each node updates its parent, children, and fragment identifier.
- Note that since each fragment has only one outgoing edge, there can at most one pair of neighboring nodes

Analysis

- The total number of phases is O(log n).
- This is because, in each phase, the total number of fragments is reduced by at least half.
- Each phase takes O(n) time.
  - All messages travel along the MST.
  - Diameter of the MST is at most O(n)
- Each phase takes O(n) messages plus the messages needed to find MOE.
- O(m + n log n) messages because in each phase a node checks its neighbor in increasing order of weight starting from the last checked node

Final Analysis

- The GHS algorithm described above correctly a distributed MST in $O(n \log n)$ rounds and uses $O(m + n \log n)$ messages.

Other Advances for MST

- Notable among other advances from the GHS algorithm are:
  - The pipelined algorithm that runs in O(n) time using O(n^2) messages.
  - The Garay – Kutten – Peleg Algorithm:
    - Uses the GHS algorithm in a clever and controlled manner
    - This controlled manner reduces the diameters of the fragments.
    - In particular, the invariant in each phase is: the number of fragments is reduced by at least a factor of two, while the diameter is not increased by more than a constant factor.
    - After a while of this controlled GHS algorithm, the diameter is within O(sqrt{n}).
      - At this point, we switch to the pipelined algorithm.