CSCI-5273 Network Systems

Final Report

Tushar Gautam (tuga2842)

Team

Tushar Gautam

Goal

Build a simple fault-tolerant distributed spanning tree of nodes for efficient data transfer from the root to all the downstream nodes. Such spanning tree topology is critical for applications such as config deployment across nodes spanning different geographical locations.

.

Architecture

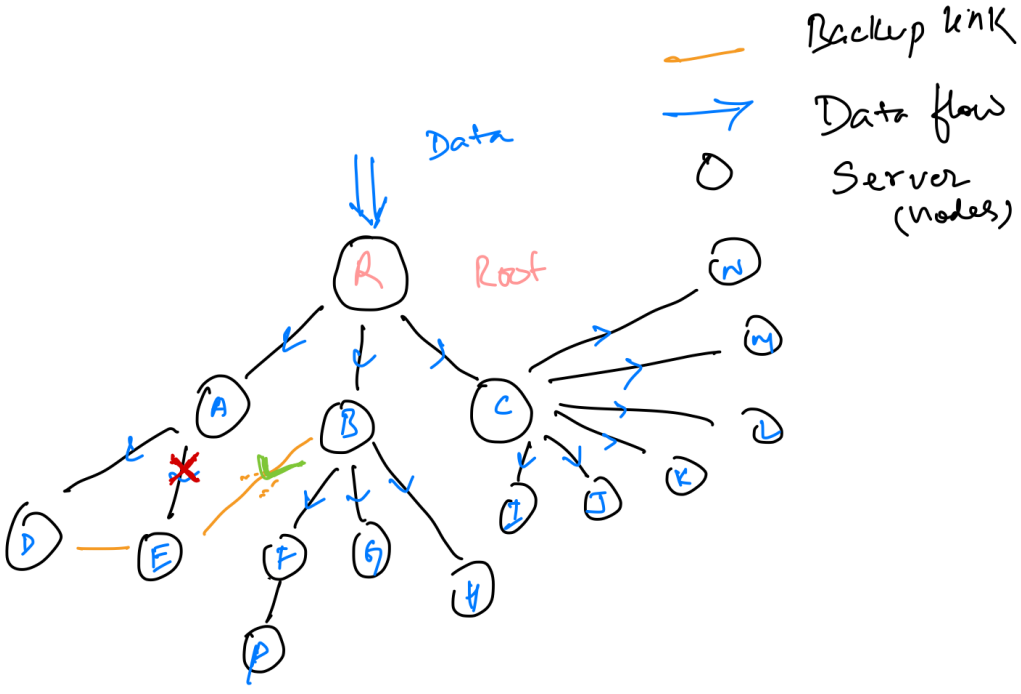


Figure 1: A rough sketch of the spanning tree over nodes

The root node **R** transmits data to its children in a hierarchical fashion. The children are at the shortest hop from Root.

Node Capacity
Each node has its capacity *C* defined which reflects the load capacity as it represents the maximum downstream connections that nodes can have. In a heterogeneous system, nodes may have different capacities.

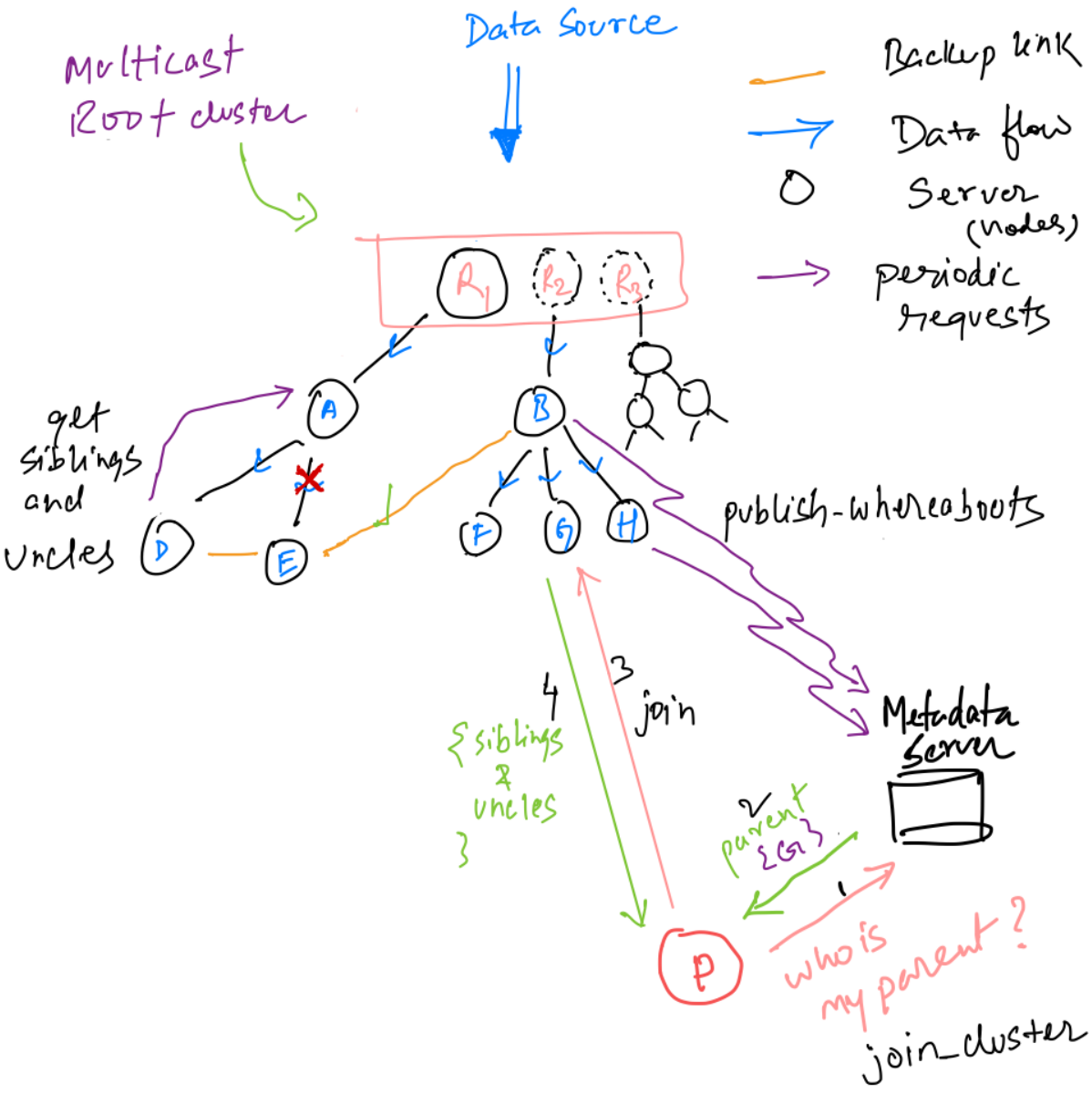
Backup link
Besides creating a primary connection to its parent, each node also maintains a membership list of its siblings $\mathbf{M_s}$ and also its parent's siblings $\mathbf{M_p}$. In case of parent link failure, the node can create a new connection to its backup node. There are several ways to decide which backup node to choose. Picking a node from a parent's sibling is preferred as it brings the node closer to the root. However, in case the parent has full capacity, the sibling nodes can be tried. This selection can be fine-tuned based on predefined weights as well. If there are multiple options to pick from the membership list, the nodes are chosen at random.
Each node should create two backup links, one to each sibling and one parent node. Further brainstorming is needed to come up with an efficient backup node selection such that all nodes serve as backup to other nodes and no nodes are oversubscribed as backup nodes.

Fault Tolerance
Backup links provide alternate options to reach to root. For eg. in Figure 1 above, node E has node A as its primary link and node D and B as its backup links. In case of failure in A, E first tries to connect via B as it places E closer to the root R and hence is preferred. In case of an internal node failure, all its children need to be distributed to their parent's siblings. To ensure there's enough capacity to handle this scenario, each node may serve only a fraction of its total capacity.

Efficient Tree Structure
For efficient tree structure, it'd be good to have a tree that is wider than very long. This ensures the height of the tree remains minimum. In a heterogeneous system, this would mean that more powerful nodes (more cores, larger bandwidth) serve at the top. The cluster can rebalance itself should a new node is added which has larger resources.
I need to brainstorm more to come up with an efficient design to support this. Likely, some metadata exchange or gossip when a new node joins would help accomplish this.

Joining the cluster
New nodes can join the cluster via the available leaf node chosen such that the leaf node farthest from the root and most loaded gets the least preference. This also ensures the tree remains balanced.

Message Flow



APIs

All intra-cluster communication happens via gRPC. Protobuf definition for parameters and APIs is available under *protos/* directory of the source code.

Data Flow APIs

Each node serves receives and forwards data via the following APIs:

- **accept_data**: This is the entry point to receive data from the parent data
- **forward_data**: To forward data to the downstream links with zero copy.

Cluster Manager APIs

A cluster manager is responsible for routine operations required for the cluster such as health reporting, peer discovery.

Health Reporting

- **publish_whereabouts**: Periodically publish { node_depth in tree, parent_node, current_children_count } to Metadata server.

Peer Discovery

- **get_siblings_and_uncles:** Periodic node requests to its parent node. The requesting node receives its siblings along with its parent's siblings i.e node's uncles.

Node Manager APIs

The node manager is responsible for maintaining a state of the node, trigger failover switch in case of parent failure.

NodeState

```
{
        children_list,
        parent_node,
        tree_depth,
        siblings_list
}
```

- **join:** Send join request to the parent node. The parent updates its NodeState and returns siblings and uncles.

- **get_new_parent:** Compute new parent based on the following priority from its local copy of NodeState:
    1. Parent's sibling
    2. Current node sibling

- **trigger_parent_switch:** Periodic monitoring of downstream data link and triggers parent switch when the current parent link breaks. Invokes *get_new_parent* followed by *join* call

Metadata Server API

The metadata server is responsible for keeping minimal whereabouts of the nodes in the tree to help initialize a new node joining the cluster. The cluster nodes send periodic updates to the metadata server. The data can persist on disk for failure recovery, however, it's perfectly alright to serve outdated data post metadata failure recovery as this outdated data will soon get updated as and when nodes send their whereabouts. Hence metadata server is pretty lightweight in terms of the number of memory footprints and requests it receives.

Key APIs are:

- **receive_whereabouts**: Receives nodes' whereabouts and stores the info in-memory cache with persistence.
- **join_cluster**: Receives request from the new node to join the cluster. Returns parent_node based on the stored whereabouts metadata. This is the initial state to join the cluster with.

Source Code

- Github repo: [tushar-rishav/CSCI5273_Final_Project]
- [Slides]