Distributed Storage System: Logical Grid Mesh

Shivam Waghela
Pranjal Sharma
Nehal Sharma
Ankita Chikodi
Shivang Mistry

CMPE 275
John Gash
December 18, 2019

Logical Grid Mesh:

We have created a 2-Dimensional Mesh structure for arranging nodes in the logical network. Each node will be connected to at most four other nodes in the network. A node will only know about its immediate neighbors and should not have any concern about other nodes in the network. In this way, we are trying to keep the network as loosely coupled and easily scalable as possible. Each node will be maintaining gRPC channels with their neighbor nodes so that the channels can be reused for all other node communications. All the nodes in the network are capable of handling requests for the addition of new nodes and the deletion of failed nodes.

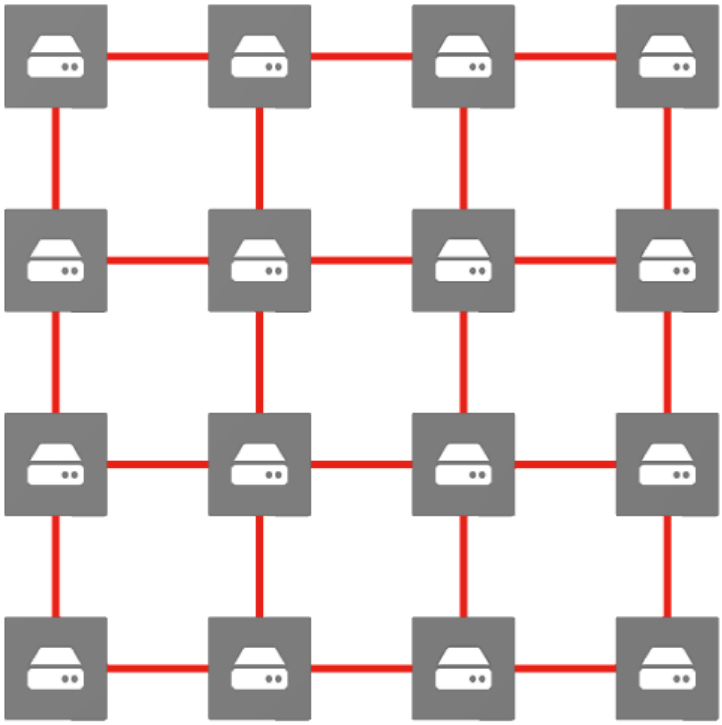

Figure: Grid mesh structure

The nodes in the network are assigned x and y coordinates based on the 2D cartesian coordinates system. The nodes can be uniquely identified through their coordinates.

Grid Mesh Creation:

We start with a single node in the network listening for requests for new node additions to the network and after that, all the existing nodes in the network can start accepting requests for new node additions. As soon as a request from a new node arrives for joining the network to one of the nodes in the network, the node will check for its available positions/ coordinates. Based on the neighborhood availability it will find and assign position and coordinates for the new node in a manner that will maintain the compactness of the grid structure. The grid mesh has some restrictions on node additions that will prevent the network from growing arbitrarily. The restriction is that the number of rows can only be one more than the number of columns and vice-versa so that the mesh always has a compact structure.

Algorithm:

1. When a new node wants to get added to the network, it will try to greet one of the nodes in the network by invoking `SayHello` gRPC method.
2. The node that received the greeting message now has to assign a position (out of TOP, BOTTOM, LEFT and RIGHT) and coordinates to the new node.
3. The node will first find all its available and unavailable neighbor positions/coordinates by looking at the `connection_dict` in the `NodeConnections` object.
4. After that, it will try to assign a position and coordinates from the available positions in such a way that the mesh structure remains compact after the new node addition.
5. It may also try to get information about its neighbor's neighbor (if required) for assigning a new node a position and coordinates to create a compact grid structure.
6. After it has assigned a new node a position and coordinates, it will send a list of IPs to the new node if the new node has other neighbors where it has been put in the network to make additional connections with them.
7. If the new node gets a list of IPs in the `additional_connections` list, it will inform the other neighbor nodes about its existence so that they know about its new neighbor.

Animation of grid mesh creation:
https://xd.adobe.com/view/16c8c251-e8d4-4735-6fea-e27600657783-60f9/screen/4e9e9e8a-40f6-4d80-8f2a-697f449a6940/Web-1920-10?fullscreen

Maintaining channels with neighbor nodes:

As creating and closing channels is an expensive operation, we are maintaining persistent channels with the neighbor nodes for reusability and performance. Every node has a single `NodeConnections` object which manages the neighbor connection information. It uses `connection_dict` to store the connection information.

The structure of the `Connection` object:

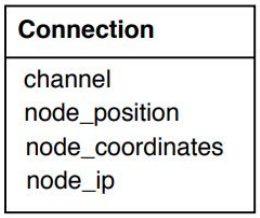

Figure: Connection Object

The `Connection` object maintains the connection information related to neighbor nodes.

The structure of `connection_dict`:

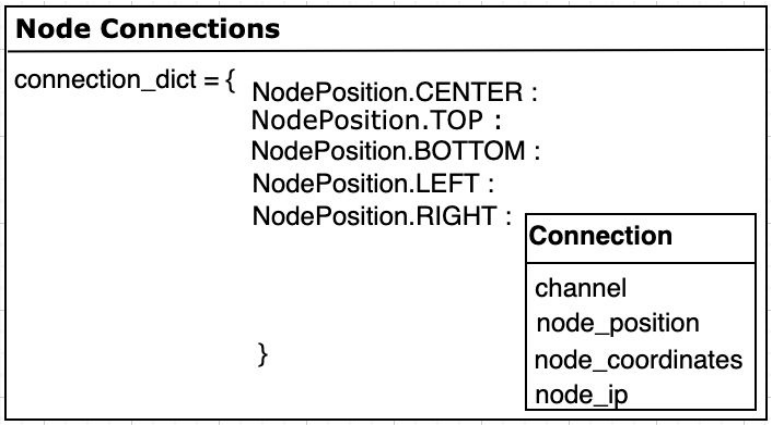

Figure: NodeConnections Object

The `NodeConnections` object maintains all the connection information of a node with other nodes in `connection_dict` dictionary.

The key in the `connection_dict` is the `NodePosition` and the value is the `Connection` object. The connection object associated with `NodePosition.CENTER` will have connection information about the node itself. The `connection_dict` will have entries only for active neighbor nodes.

Managing a shared resource:

We have a `NodeConnections` object which will be modified by both server and client threads. This object will also be used by several other threads who want to transfer data between nodes. We need to make sure that this shared resource is managed properly so it will always be in a consistent and stable state.

We only want a single thread to be modifying the object at a time. For that, we have applied locks to the critical section of the code where the `NodeConnections` object gets modified. Locks are used in `NodeConnections.add_connection` and `NodeConnections.remove_connection` methods.

Code Organization & Structure:

Writing maintainable, reusable and scalable code is very important for the project of such a large scale where our codebase will be used by several teams to integrate their solutions. We went through several iterations to improve our code and make it more maintainable and reusable. Following are the different iterations of our codebase which can also be found on our GitHub repository (https://github.com/shivamwaghela/DistributedStorageSystem/) under releases section:

- v1.0-beta
- v2.0-beta
- v2.0
- v2.1
- v2.2 (Latest Logical Grid Mesh Code)

Initial Version

Our initial version of the code had two different processes - client and server process. A single process cannot be used for both serving and sending the requests as the main

thread will be blocked when it starts listening for requests (unless we use multithreading). But we did not had any multithreading initially so we landed with two processes. The client process was used to send data or communicate with other nodes and the server process listening for data.

For sharing information between both the processes we used a shared file which we found out was causing lots of degradation in system performance as there were too many file I/O operations. Also maintaining two processes would increase system resource utilization as compared to just a single process.

Worst of all was that now every team's solution will run as a separate process which will not only consume a lot of resources but would also put a significant load on the shared file resource.

Latest Version

Several lessons were learned during our early implementations and we decided to address them to improve our system performance and maintenance.

Several design goals were laid out for our new implementation:

- Use multi-threading instead of separate processes
- Remove shared file between processes/ threads
- Make code more maintainable and reusable
- Improve logging by introducing logging at different levels
- Separate out functionality into several manageable and reusable modules

We combined both client and server processes into a single process with multiple threads. Now any additional functionalities like replication, node traversal, heartbeats will run as separate threads.

We also separated out different functionalities into smaller, manageable modules so that any team that wants to integrate with mesh will just have to add their modules to the `node` directory and start their thread in the `node.py` file. With this code reorganization, we were seamlessly able to integrate other team's features without disrupting the existing features and functionalities.

The use of the logging module significantly helped in debugging our code as we were able to log messages at different levels based on their severity.

Pulse Module:

Nodes in the large distributed systems go down frequently due to irrecoverable exception in the code, network failure, disk failure, memory failure, power outages, etc. The network has to be resilient to these failures and should always remain in a stable and consistent state. With a large distributed system we should always expect failure and provide mechanisms to handle it smoothly.

As we are responsible for building up the mesh structure, we also have the responsibility to detect and handle node failures in the mesh. We are using two mechanisms for detecting node failures.

We declare the node to be down if:

- It is physically disconnected from the network
- The node process is down

We use ping to test for the physical connection and grpc.channel_ready_future for ensuring that there is channel connection with the node and that the node process is active. After we are sure that the node is down, we remove it from our active connections.

Logging:

There are several levels of logging done in code. The information logging messages are logged using `logger.info`, the error logging messages are logged using the `logger.error` and debugging messages are logged using `logger.debug`.

By separating logging messages into different categories helps us analyze the state of the system properly. And most importantly we can significantly reduce down the logs by turning off certain types of log messages.

For example, after the system is fully tested and running we can set the logging level to logging.INFO and it will silence all logging below the level. As logging.DEBUG is below that level, we won't see debug logs and with that, we can significantly reduce down the log I/O and increase system performance.

Machine Information Module:

We introduced a small module that will help us get the machine's node IPv4 address, CPU usage, and available system memory details.

Getting the IPv4 address from the machine was convenient than hardcoding the IPv4 address somewhere in the code or some config files.

Pros of Grid Mesh Design:

- The network is loosely coupled.
- The network will adapt itself accordingly to the addition or deletion of nodes to maintain the compact grid structure.
- Nodes maintain gRPC channels with neighbors for channel reusability.
- All nodes are capable of handling requests for the addition of new nodes.
- Maintaining direct connections only with their neighbor nodes.
- Unlike fully connected mesh topology, the 2D Mesh network highly reduces the overhead on each node as it has to maintain connections with at most 4 nodes.
- As all the nodes are arranged in a grid and have cartesian coordinates, it is easy to locate and find the shortest path based on simple math calculation.
- The structure can be extended to create a 3D mesh with minimum code changes.

Cons of Grid Mesh Design:

- Since each node is not connected to every other node in the network, the time to forward data to a particular node increases.

Other Designs Considered and Tested:

Fully Connected Mesh

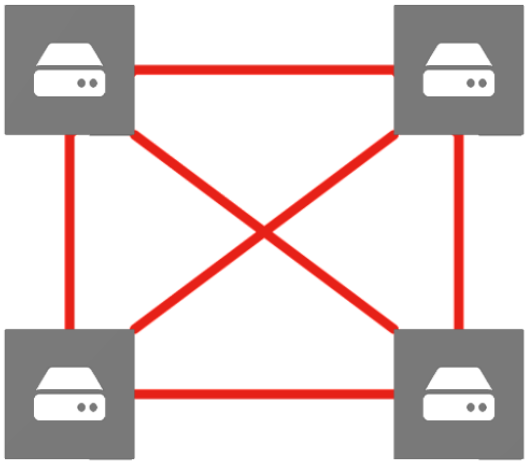

Figure: Fully Connected Mesh

Pros:

- Direct connections with each node
- Low communication latency
- Easy to implement

Cons:

- Too much noise and communication overhead for just maintaining the network
- High resource utilization in establishing connections when a new node is added. For every new node added to the network, it has to establish n-1 connections to get connected to everyone (where n is the size of the network).
- Heartbeats also significantly increase the noise as every node will send n-1 heartbeats periodically.

Overall Experience:

As we were the team who were responsible for building the underlying mesh network we had to make sure that it was robust, scalable and resilient to failures. We cannot have any room for error as other teams that will be working on top of our network will face problems if the mesh fails to provide the services as per the contract.

Fortunately, we were able to test our mesh codebase thoroughly as we had multiple teams working on top of it so we were able to resolve any bugs or performance issues as they were encountered.

Integration with the node traversal, memory, middleware, and the application team went smoothly as we were integrating and fixing issues constantly throughout the course. We were successfully able to test end to end with node traversal, memory, middleware, and the dropbox application team. We are able to receive data from the application and were able to store it in the network efficiently. Also, we were able to handle read requests from any node in the network.

We think we had issues in the class demonstrating our solution as there was a last-minute merge on our stable and fully tested branch from the replication and the gossip team. As it was last-minute we were not able to test and fix issues from the merging of the branches. With this kind of experience, we learned the importance of freezing of the codebase before any kind of demonstration or product launch.

We were able to meet our design expectations that were set out at the beginning of the project. Also, we were able to significantly improve our design and implementation based on the regular feedback received from Prof. Gash.

We think a smaller team size would lead to a better experience in integration as we wouldn't have a lot of inter-dependencies. If the class is of size 40 then we can have two teams of size 20. This would help in better collaboration and success of the project.

Most importantly we are very grateful to Prof. Gash for providing valuable feedback and guidance throughout the course so we can constantly try to push ourselves to make better design and implementation decisions for our distributed storage system.

Source Code:
https://github.com/shivamwaghela/DistributedStorageSystem/tree/dev-grid-mesh