Data Replication

Reverse Lecture-3 + Final Project Report

CMPE-275

SAN JOSÉ STATE UNIVERSITY

Submitted to
Professor John Gash

By
Mudambi Seshadri Srinivas
Vaishali Koul
Roberto Campbell
Tin Vu
Phuong Tran

Table of Contents

Abstract

This document provides a detailed explanation of the data replication part in the Distributed File Storage System. In a distributed system, data replication ensures reliability and availability among nodes: if one or more servers fail, the system can still function properly even though the node which contains the write is not available. The performance of the system can also be improved due to lower access time and higher scalability; we can reduce response time as the data can be acquired locally, and the system can scale better as requests can be serviced by multiple servers instead of one.

Reason for a three-phase replication :

The initial idea was having the replicas in the best suited neighboring nodes. However, this could create hotspots in the system which might lead to complete data loss if the path to the node is blocked, or if there is a network partition in the system. Selecting random nodes spread across the system solves that problem. Also, since the read requests were being forwarded in a random fashion by the chunk server, the probability of hitting the right node that has the value is higher. Thus, spreading the data replicas to further nodes allows faster reads.

Replication in the 2D Mesh File storage system is held in three phases :

1. **Gossip About Gossip:**

   Reason: This is not a Master-Slave architecture with a single node dependency for accepting the writes. Also, in the 2D Mesh architecture, all nodes are not connected to others in the system, gossiping was the best way for the team to pass information.

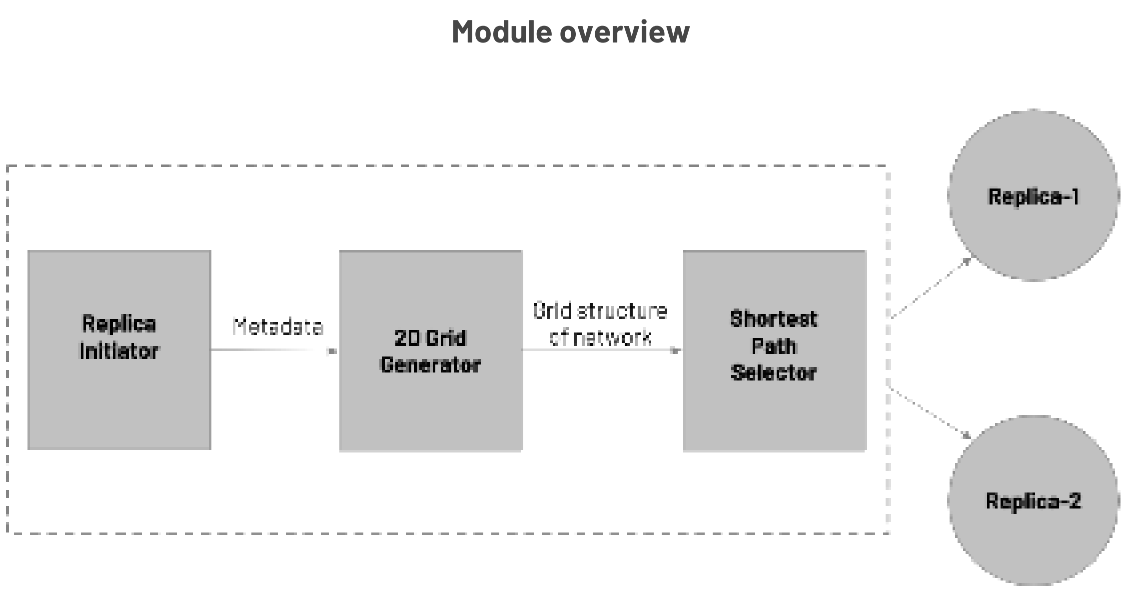
   Idea: In Gossip about Gossip, not only a random node is selected to pass some information but the information being passed is the history of the Gossip itself. A node not only passes the value that it knows but also passes what it heard from others.

2. **Virtual Voting :**
   A node after receiving a value will compute locally to find the best node to replicate in the system. The computation is a comparison of the already existing local value and the newly received value. The node that receives a request (write/update/delete) initiates a gossip where it compares the capacities of its neighboring nodes and the broadcasts the lowest-capacity node that can handle the data. A convergence criteria of 50% of the nodes agreeing to a value stops the gossip and gives the initiator a node value to

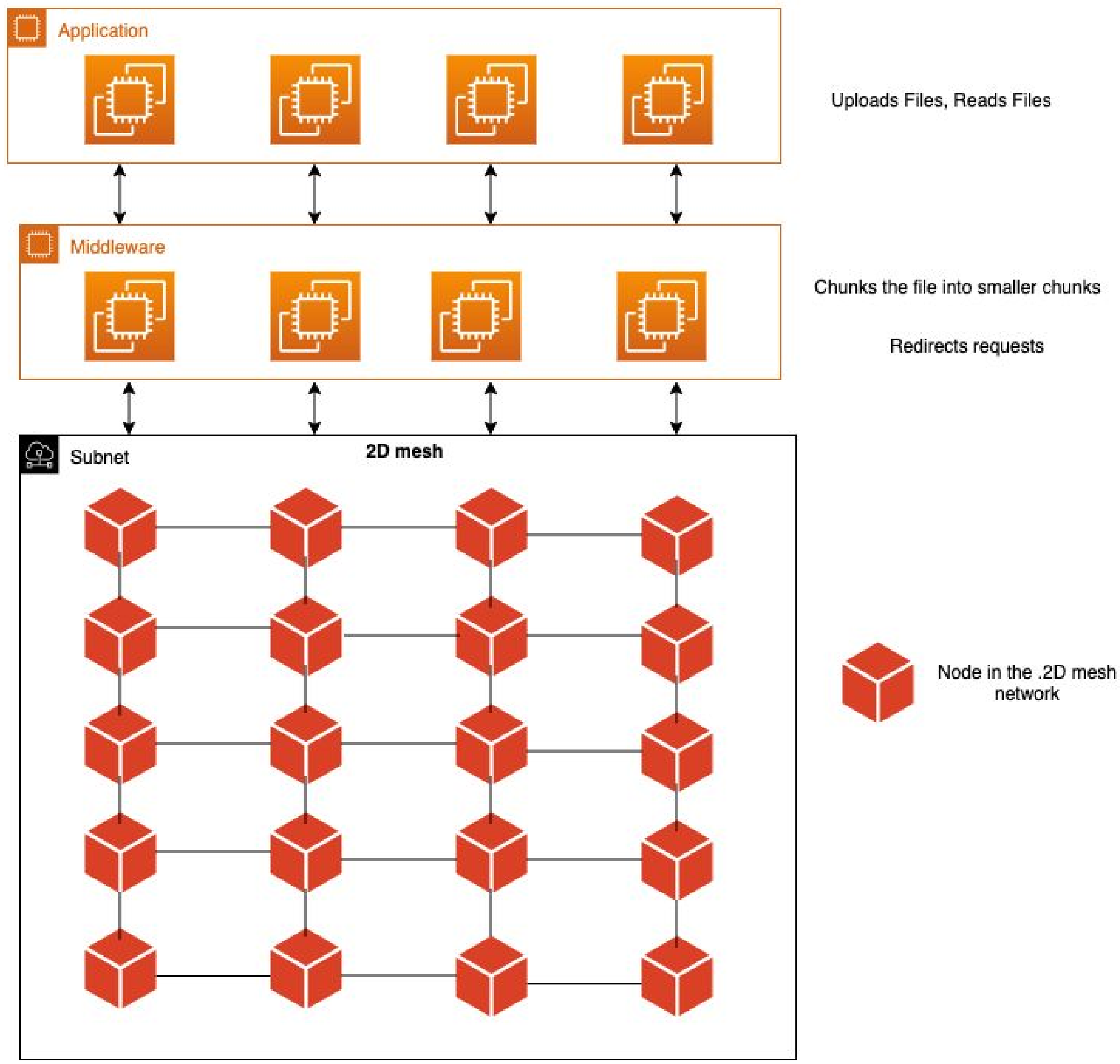
replicate. When a node receives the same value more than 10 times, it gets added to the Blacklist and stops gossiping.

3. **Creation of a logical snapshot of the network**
   After receiving the coordinates of the alive nodes, the replication initiator creates a logical snapshot of the grid that helps it traverse to the node it wants to replicate at.

4. **Traversal**
   A Breadth-first traversal leads the data to the soon to be a replica node. The best thing about this implementation was the shortest path calculation with failure detection. That ensures that the initiating replica receives an acknowledgment from the replicating server.



System Architecture



Architecture Overview

Technology Stack and Methodologies Used

Python	
gPRC	
Protobuf	
Virtual Consensus	
Python-Socket	

Module Functionalities

- Virtual Consensus (Gossip of Gossip)
- Generation of a logical snapshot of the network
- Calculation of best path from source to destination using a Breadth-First Search Algorithm
- Establishing gRPC channels for data forwarding
- Implementing Vector clocks for consistency

Implementation Overview

Stage - 1: When a “**write request**” or an “**update request**” is performed, middleware redirects the request to one of the nodes in the underlying architecture.

Let us assume **Node - p** gets the request.

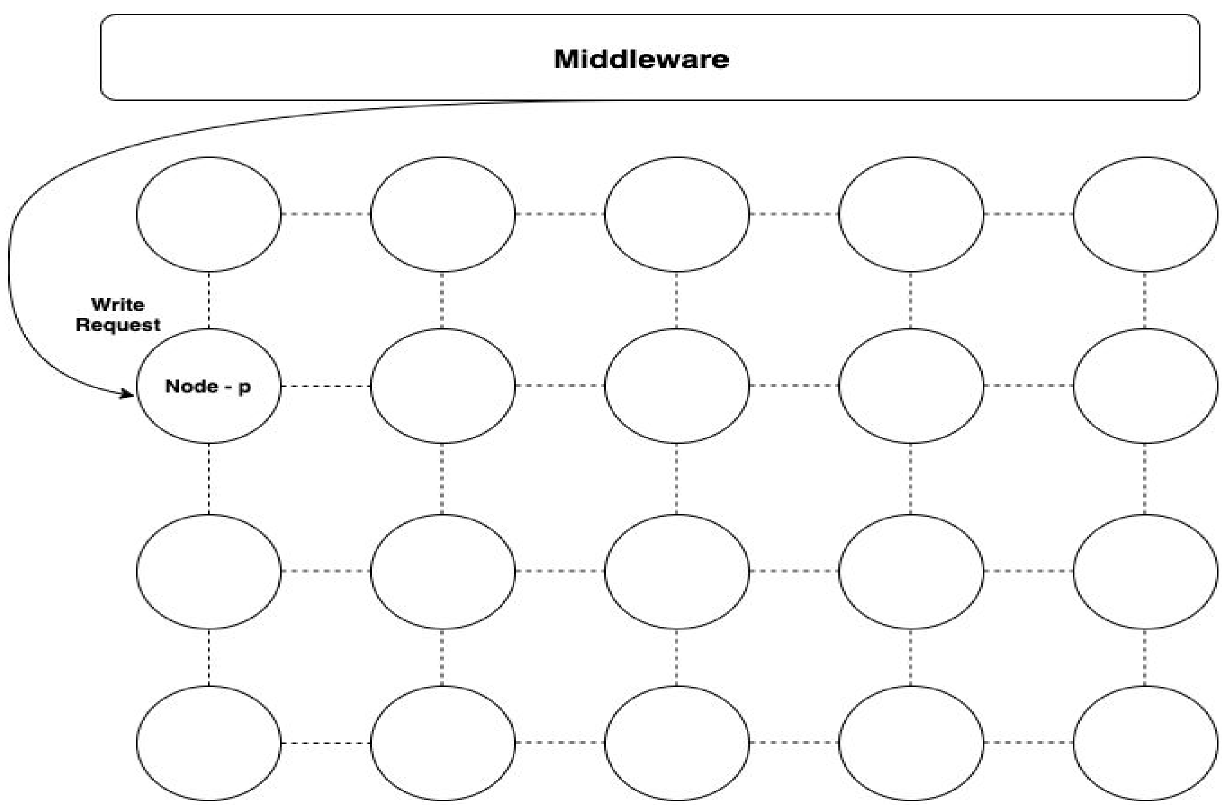


Figure. The initial state of the system

Stage - 2: On **Node - p** “**Best nodes for replication**” module gets triggered for picking the best nodes to perform replication which are spread across the network. This method avoids the hotspots around the nodes that have taken the “**write requests**”. This methodology also helps in “**read requests**”. When a random node receives a read request, there are higher chances of nodes (holding the required data) encountered in a quick time when there is a spread of replica nodes.

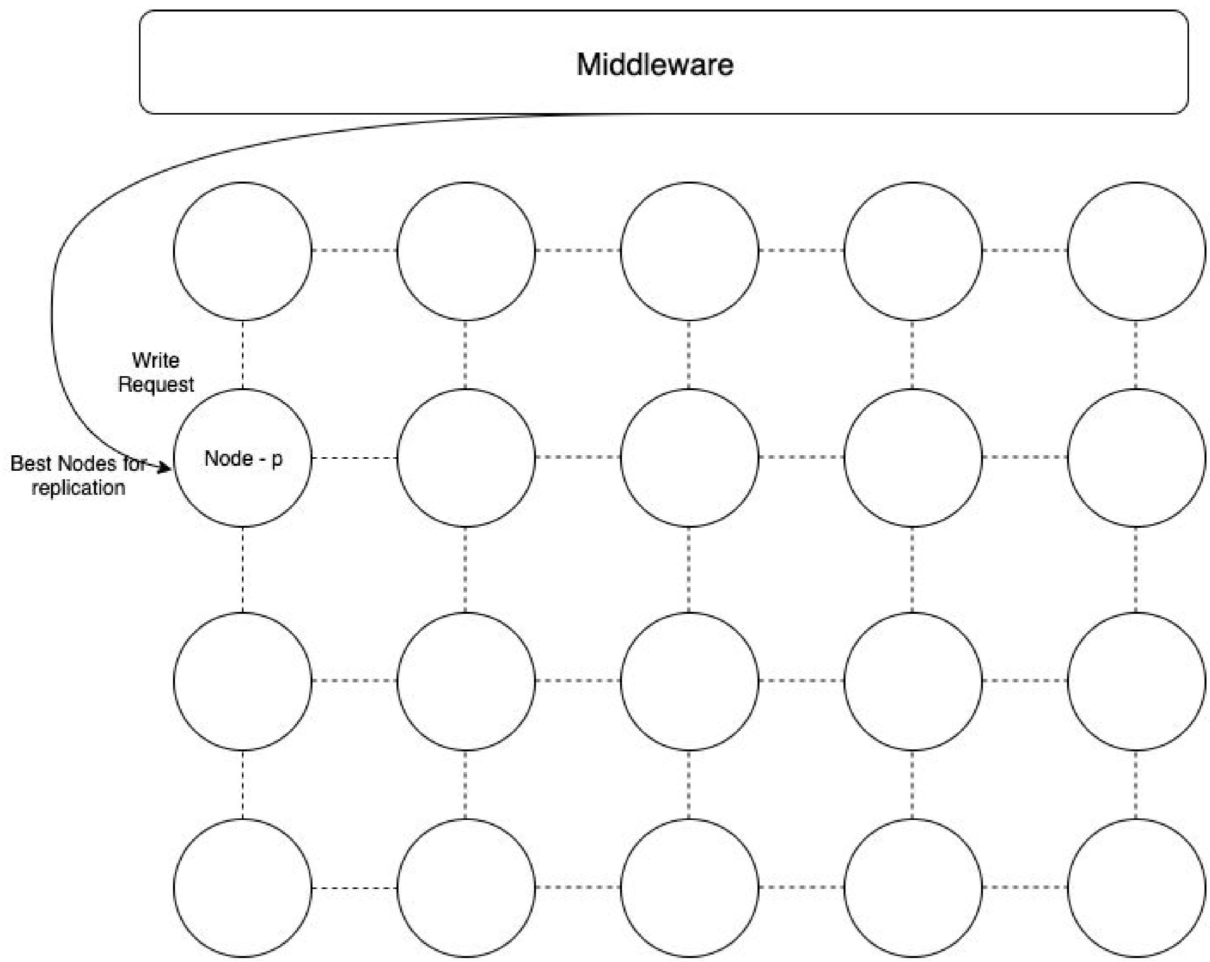
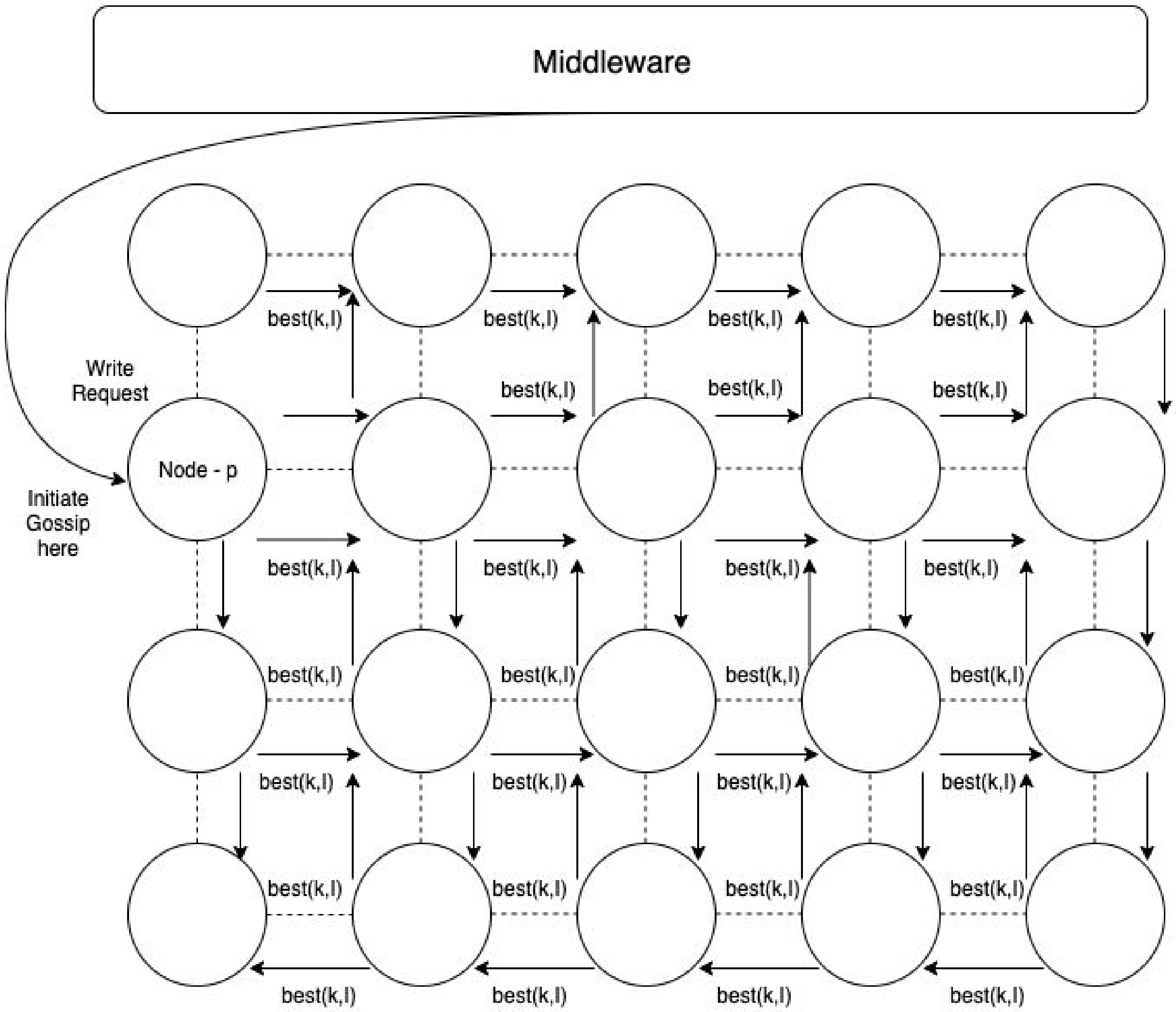


Figure. Trigger “Best nodes for replication” for picking 2 nodes

Stage - 3: Gossip of Gossip is triggered to find the best nodes. In every gossip call, three different evaluations are performed:

1. Check for convergence
2. Calculate the best capacity of neighbors
3. Evaluate Incoming best capacity with the local best capacity

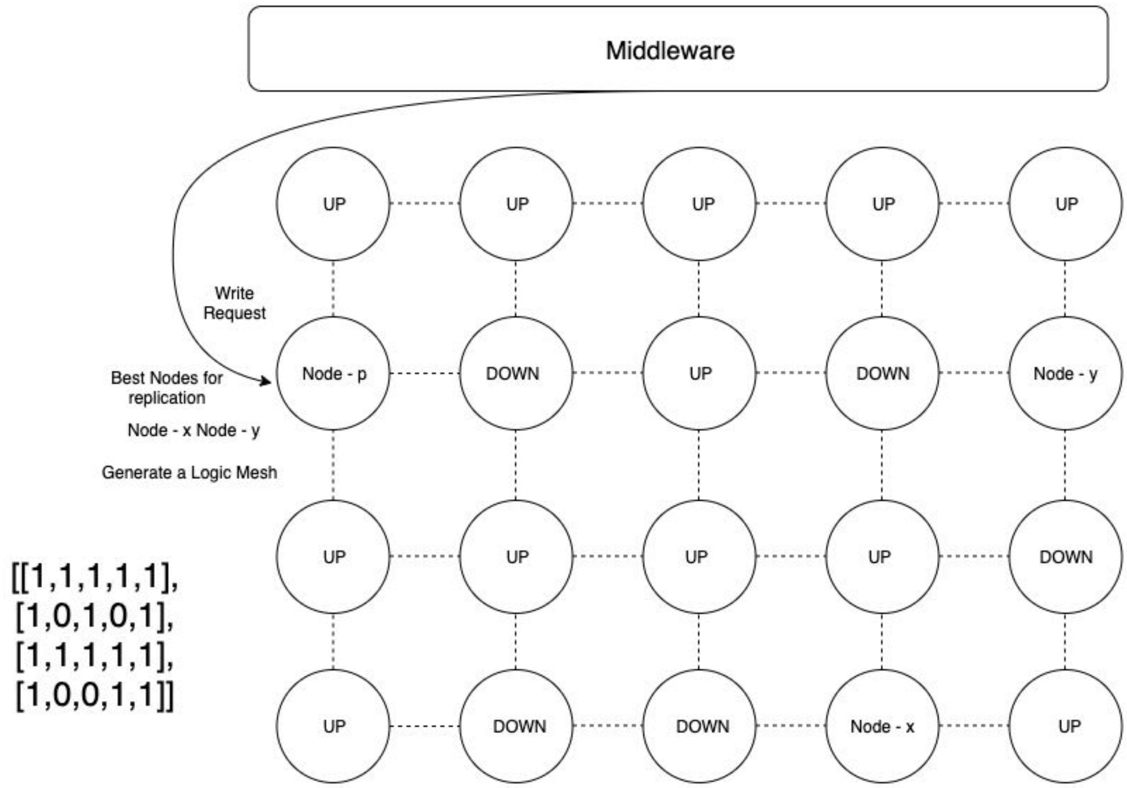
Gossip the evaluations accordingly to all the neighbors
This is performed until the convergence is met.



Gossip of Gossip Convergence

K - Best Known Node Stats
I - Incoming Node Stats

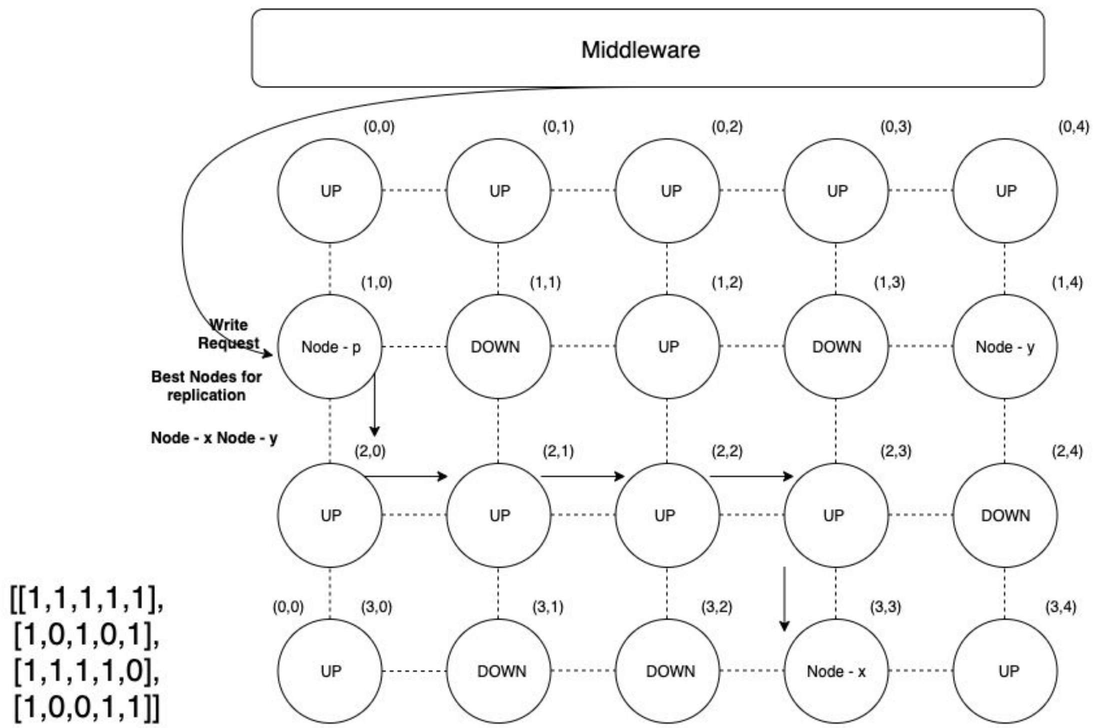
Stage - 4: Based on the metadata that gets exchanged on a network level, we build a logical snapshot of the network for finding a path to the agreed nodes for replication. Assume node-x and node-y are picked as the best nodes for the replication. As shown in the matrix if a node is down we mark it as 0 and if it is up we mark it as 1 and create a logical mesh out of the available data.



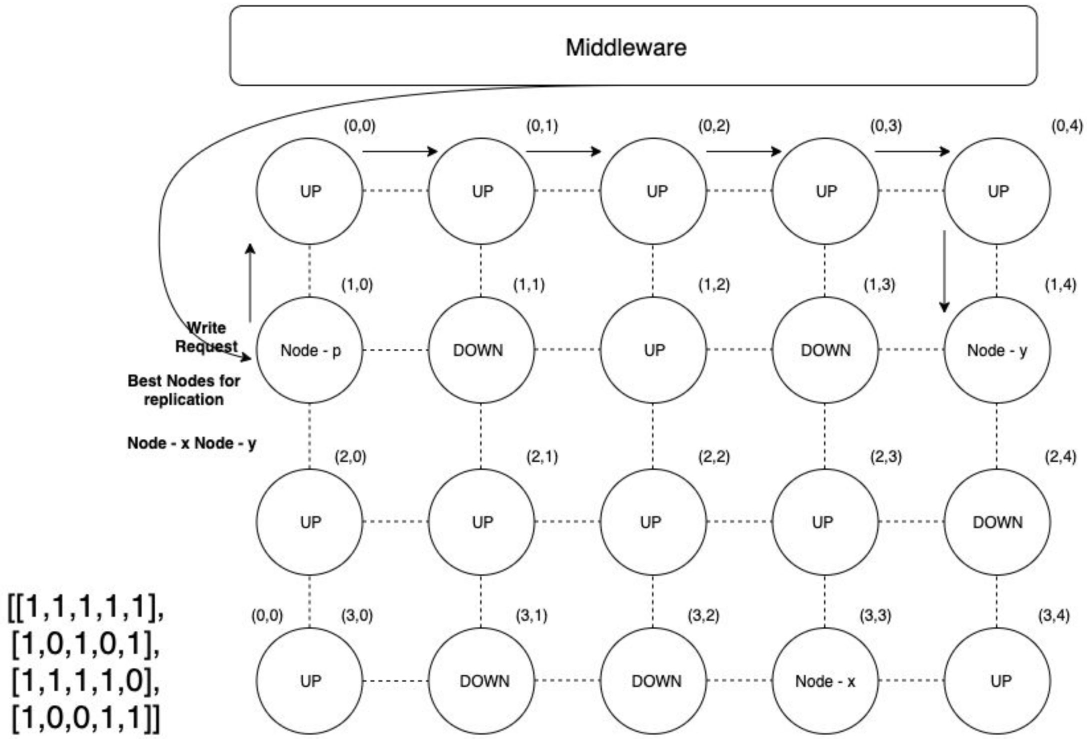
Stage - 5: After performing "Gossip of Gossip" to achieve virtual consensus and generating a logical snapshot of the underlying network, we now calculate the shortest path with failure nodes from the replication initiation node (node - p) to destination replica-nodes (node-x and node-y). All these calculations are performed on the node that has initiated the replication.

Stage - 6: Once the shortest path gets established. Data has to be packed into objects and passed on to the neighbor nodes listed in the path. Every time a node gets the data to be replicated, two operations are performed

If the node is the destination node, it triggers the upload function which writes to the memory
If the node is not the destination node, it evaluates the forwared_to node using the metadata and passes on the data object until step 1 is satisfied.



Best Path to Node-x = [(1,0),(2,0),(2,1),(2,2),(2,3),(3,3)]



Best Path to Node-y = [(1,0),(0,0),(0,1),(0,2),(0,3),(0,4),(1,4)]

Data is replicated successfully on two nodes.

Implementation

Data Replication pseudo-code

<table>
<tr>
<td>Virtual Consensus</td>
<td>

```
# Gossip Receive
def receive_gossip(self, gossip_received):
       check_for_convergence = convergence(gossip_received)
       if check_for_convergence == False:
         best_known = max(capacities_of_neigbors)
         best_capacity_node = max(gossip_received, best_known)
         transmit_gossip(best_capacity_node)
       else:
         #wait for other gossip

# Gossip Transmit
 def transmit_gossip(self, gossip_trasmit):
       all_neighbors = fetch_neighbors(self.IPaddress)
       transmit(all_neighbors)

# Gossip Convergence Check
def convergence(self, gossip_received):
 if gossip_received == gossip_known:
   counter=counter+1
 if counter == 10:
   black_listed_nodes.append(self.IPaddress)
 # optimization
   all_neighbors = fetch_neighbors(self.IPaddress)
   black_listed_nodes.append(all_neighbors)
   if len(black_listed_nodes) >= 0.5 *  (total_network_length):
   return True
 return False
```

</td>
</tr>
</table>

Generation of a logical mesh from the available metadata	<pre>create_2D_grid(self, metadata): metadata = sort_list_on_y(metadata) metadata = sort_list_on_x(metadata) dictionary = {} map each element in dictionary to a key number_of_rows = absolute(min(x)) + max(x) number_of_cols = absolute(min(y)) + max(y) list = reshape(metadata, number_of_rows, number_of_cols) call_bfs(list)</pre>
Breadth-First Search for shortest path	<pre>bfs(grid,start_node, target_node) queue: to append the path set: to keep track of visited nodes Append start_node to the queue Iterate until the queue is Empty path = Pop from queue last_ele = last element(node) in path if last_ele of path equal to target node return path else iterate over every neighboring element of last_ele in the path if neighboring element not in set append the neighboring element to the path and queue add the neighboring element to the set</pre>

Data Replication code flow

Method	Code Snippet
Main function	<pre># in the main function of the program Gossip of Gossip runs as a background thread if __name__ == "__main__":</pre>

	`` <code> gossip_thread = GossipProtocol() # initiate gossip thread running as a background process gossip_thread.start_threads() <code> ```
Upon “write” request	``` def upload_chunk_stream(self, request_iterator, context): ''' Write to memory upon request from middleware Initiate Replication module ''' # Wite on to the memory success = self.memory_manager.put_data(request_iterator, hash_id, chunk_size, number_of_chunks, False) # Download content of file for replication from the memory message_stream_of_chunk_bytes = self.memory_manager.get_data(hash_id) # Initiate the replication module as a thread by passing chunks Thread(target=self.replicate_data, args=(message_stream_of_chunk_bytes, replMetadata)).start() return storage_pb2.ResponseBoolean(success=success) ```
Starting of thread initiates the below function	``` def replicate_data(self, message_stream_of_chunk_bytes, metadata): ''' Trigger Virtual Consensus Create a logical snapshot of the network Calculate the best paths for replica nodes Replicate data ''' replication_flag = True while True: # initiates the replication procedure start_replica() # wait time for gossip gconvergence (usually in milliseconds) time.sleep(5) ```

```
        # capture the list of nodes for replication from the global
object
        nodes = globals.nodes_for_replication
        # If found 2 nodes for replication
        if len(nodes) > 2:
            # calculate the path for first replica by passing on
metadata of the network
            path_one = get_best_path(globals.whole_mesh_dict,
nodes[0])
            status_one = replication(path_one,
message_stream_of_chunk_bytes, metadata)
            # calculate the path for second replica by passing on
metadata of the network
            path_two = get_best_path(globals.whole_mesh_dict,
nodes[1])
            status_two = replication(path_two,
message_stream_of_chunk_bytes, metadata)
            print("First Replication Status  : ", status_one)
            print("Second Replication Status : ", status_two)
            if status_one and status_two :
                break
        # If found only one best node  for replication
        elif len(nodes) > 0:
            path_one = get_best_path(globals.whole_mesh_dict,
nodes[0])
            status_one = replication(path_one,
message_stream_of_chunk_bytes, metadata)
            if status_one:
                break
        # if no nodes are found for replication
        else:
            print("No nodes for replication")
# method that initiates the replication module
def start_replica():
   '''
       Establishes a UDP socket connection for the exchange of data
amongst neighbors
```

	`` Triggers Gossip of Gossip algorithm which runs as a thread in the background ''' serverAddressPort = (globals.my_ip, 21000) UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM) dict = {} # create an object with the following parameters message = json.dumps({"IPaddress": globals.my_ip, "gossip": False, "Dictionary": dict, "BlackListedNodes": []}) UDPClientSocket.sendto(message.encode(), serverAddressPort) ```

Gossip about Gossip

Method	Code Snippet
Receive Message	``` # Receive message - gossip of gossip ''' Receive a message from the neighbor nodes Check for convergence If convergence is met stop gossiping and edit the global value for best nodes Else evaluate the best capacity nodes Gossip the evaluation results ''' def receive_message(self): while True: message_Received, address = self.UDPServerSocket.recvfrom(1024) data = json.loads(messageReceived.decode()) IPaddress = data.get("IPaddress") gossip_flag = data.get("gossip") Dictionary = data.get("Dictionary") BlackListedNodes = data.get("BlackListedNodes") ```

```
        # if blacklisted nodes is greater than half of the nodes in
the network
        # Convergence is met
        if len(BlackListedNodes) >= 0.5 * totalNodes:
            best_ip_addresses = list(Dictionary.keys())
            if globals.nodes_for_replication == None:
                globals.nodes_for_replication = []
                globals.nodes_for_replication.append(best_ip_addresses)
            else:
                globals.nodes_for_replication.append(best_ip_addresses)
        continue

        # This part is triggered only for once during the initiation
of replication
        if str(IPaddress) == self.IPaddress and gossip_flag == False:
            self.blacklisted_nodes=[]
            list_of_neighbors = self.fetch_all_neighbors()
            # get minimum capacity used neigbors
            minimum_capacity_neighbor_one,
minimum_capacity_neighbor_two =
self.get_minimum_capacity_neighbors(IPaddress)

        if minimum_capacity_neighbor_one==None and
minimum_capacity_neighbor_two== None:
            continue

        max_size = sys.maxsize
        # find two of the minimum of capacity used nodes
        minimum_capacity_one = min(minimum_capacity_neighbor_one[1],
max_size)
        minimum_capacity_two = min(minimum_capacity_neighbor_two[1],
max_size)

        self.counter = 1

        IPaddress, gossip, Dictionary =
self.updated_message_util(data, minimum_capacity_one,
```

	`` minimum_capacity_two, minimum_capacity_neighbor_one[0], minimum_capacity_neighbor_two[0], True) for ip in range(len(list_of_neighbors)): response = os.system("ping -c 1 " + list_of_neighbors[ip].strip('\n')) # If the node is down if response == 0: self.transmit_message(IPaddressOne, IPaddress, False, Dictionary, BlackListedNodes) else: # If the node is up continue elif gossip_flag: Convergence_Value = self.checkforConvergence(data.get("Dictionary"), BlackListedNodes, address[0]) if Convergence_Value == True: continue else: list_of_neighbors = self.fetch_all_neighbors() # Calculate the best available capacity of neighbors minimum_capacity_neighbor_one, minimum_capacity_neighbor_two = self.get_minimum_capacity_neighbors(IPaddress) Temp = data.get("Dictionary") if minimum_capacity_neighbor_one==None and minimum_capacity_neighbor_two== None: continue Local_Dict = {minimum_capacity_neighbor_one[0]:minimum_capacity_neighbor_one[1] , minimum_capacity_neighbor_two[0]:minimum_capacity_neighbor_two[1]} Temp.update(Local_Dict) New_Dict = {} New_Dict.update(sorted(Temp.items(), key=lambda x: x[1])) first_minimum = New_Dict[list(New_Dict.keys())[0]] ```

```
      second_minimum = New_Dict[list(New_Dict.keys())[1]]

      Temp_Dict =
{list(New_Dict.keys())[0]:first_minimum,list(New_Dict.keys())[1]:sec
ond_minimum}

       # If the local data is updated
      if Temp_Dict != Local_Dict:
         IPaddress, gossip, Dictionary_updated =
self.updated_message_util(data, first_minimum, second_minimum,
list(New_Dict.keys())[0],
list(New_Dict.keys())[1], True)

       # broadcast the updated information to the neighbors
       for ip in range(len(list_of_neighbors)):
           response = os.system("ping -c 1 " +
list_of_neighbors[ip].strip('\n'))
            if response == 0:
               IPaddressOne = list_of_neighbors[ip].strip('\n')
               self.transmit_message(IPaddressOne, IPaddress, True,
Dictionary_updated, self.blacklisted_nodes)
           else:
                continue
       else:
            # broadcast the updated information to the neighbors
           for ip in range(len(list_of_neighbors)):
             response = os.system("ping -c 1 " +
list_of_neighbors[ip].strip('\n'))
             if response == 0:
                IPaddressOne = list_of_neighbors[ip].strip('\n'),
Dictionary = data.get("Dictionary"),
self.transmit_message(IPaddressOne, IPaddress, True, Dictionary,
self.blacklisted_nodes)
              else:
                  continue

    # If convergence is not met
```

```
    elif gossip_flag == False and self.IPaddress != IPaddress:
       self.blacklisted_nodes = []
       list_of_neighbors = self.fetch_all_neighbors()
       # Calculate the best available capacity of neighbors
       minimum_capacity_neighbor_one, minimum_capacity_neighbor_two
= self.get_minimum_capacity_neighbors(IPaddress)
       Temp = data.get("Dictionary")

      if minimum_capacity_neighbor_one==None and
minimum_capacity_neighbor_two== None:
        continue

      Local_Dict =
{minimum_capacity_neighbor_one[0]:minimum_capacity_neighbor_one[1],
minimum_capacity_neighbor_two[0]: minimum_capacity_neighbor_two[1]}
      Temp_Dict.update(Local_Dict)
      New_Dict = {}

      New_Dict.update(sorted(Temp.items(), key = lambda x : x[1]))
      first_minimum = New_Dict[list(New_Dict.keys())[0]]
      second_minimum = New_Dict[list(New_Dict.keys())[1]]
      Temp_Dict = {list(New_Dict.keys())[0]:first_minimum,
list(New_Dict.keys())[1]:second_minimum }

      if Temp_Dict != Local_Dict:
        IPaddress, gossip, Dictionary_updated =
self.updated_message_util(data, first_minimum, second_minimum,
list(New_Dict.keys())[0],
list(New_Dict.keys())[1], True)
       # broadcast the updated information to the neigbors
       for ip in range(len(list_of_neighbors)):
         response = os.system("ping -c 1 " +
list_of_neighbors[ip].strip('\n'))
         if response == 0:
            IPaddressOne = list_of_neighbors[ip].strip('\n')
            self.transmit_message(IPaddressOne,s IPaddress, True,
Dictionary_updated, self.blacklisted_nodes)
```

	<pre> else: continue Else: # broadcast the updated information to the neigbors for ip in range(len(list_of_neighbors)): response = os.system("ping -c 1 " + list_of_neighbors[ip].strip('\n')) if response == 0: IPaddressOne = list_of_neighbors[ip].strip('\n') Dictionary = data.get("Dictionary") self.transmit_message(IPaddressOne, IPaddress, True, Dictionary, self.blacklisted_nodes) else: continue</pre>
Transmit Message	<pre>def transmit_message(self, hostname, IPaddress, gossip, Dictionary,BlackListedNodes): ''' Gossip the updated and evaluated information ''' serverAddressPort = (hostname, 21000) bufferSize = 1024 message = json.dumps({"IPaddress": IPaddress, "gossip": gossip, "Dictionary": Dictionary, "BlackListedNodes": self.blacklisted_nodes}) self.UDPServerSocket.sendto(message.encode(), serverAddressPort)</pre>
Convergence Criteria Function	<pre>def checkforConvergence(self, Dictionary, BlackListedNodes, address): ''' Validate local message with the incoming message Evaluate the count of consecutive appearance of the same message</pre>

```
        If more than 10 add self to blacklisted nodes
        If the length of blacklisted nodes > 50 % of nodes in the
network return True
   '''
        message_received = Dictionary
        # If no nodes are blacklisted
        if BlackListedNodes == None:
            #if the same message is received, blacklist self
            if self.local_message == message_received:
                self.counter += 1
                if self.counter >= 10:
                    BlackListedNodes = []
                    BlackListedNodes.append(self.IPaddress)
                    BlackListedNodes = set(BlackListedNodes)
                    listofNeighbors = self.fetch_all_neighbors()
                    self.blacklisted_nodes = self.blacklisted_nodes +
BlackListedNodes
                    self.blacklisted_nodes =
set(self.blacklisted_nodes)
                    self.blacklisted_nodes =
list(self.blacklisted_nodes)
                    if len(self.blacklisted_nodes) >= 0.5 *
totalNodes:
                        self.counter = 1
                        return True
                return False
            else:
                #If message received is different
                self.local_message = {}
                self.local_message = message_received.copy()
                self.counter = 1
                return False

         # If few nodes are already blacklisted
         if BlackListedNodes != None:
            if self.local_message == message_received:
                self.counter += 1
```

<table>
<tr><td></td><td>

```python
        if self.counter >= 10:
            if self.IPaddress not in BlackListedNodes:
                BlackListedNodes.append(self.IPaddress)
            self.blacklisted_nodes = self.blacklisted_nodes + BlackListedNodes
            self.blacklisted_nodes = set(self.blacklisted_nodes)
            self.blacklisted_nodes = list(self.blacklisted_nodes)
            if len(self.blacklisted_nodes) >= 3:
                self.counter = 1
                return True
        return False
    else:
        self.local_message = {}
        self.local_message=message_received.copy()
        self.counter = 1
        return False
```

</td></tr>
</table>

Write to Replicas

<table>
<tr><th>Method</th><th>Code Snippet</th></tr>
<tr><td>Best Path Calculation</td><td>

```python
# Function to create a 2-D logical snapshot of the network and perform BFS to calculate the shortest path
def get_best_path(whole_mesh_dict, destination_ipaddress):
    '''
        Capture metadata from network
        Fill gaps in the network
        Reshape to a 2D representation of the network with node failures
        Perform BFS with Source and Destination and return PATH
    '''
```

</td></tr>
</table>

```python
# initialize original list from metadata of the network
original_list = list(whole_mesh_dict.keys())
my_list = list(whole_mesh_dict.keys())

max_rows = -sys.maxsize - 1
for item in range(len(my_list)):
    if my_list[item][0] > max_rows:
        max_rows = my_list[item][0]

max_cols = -sys.maxsize - 1
for item in range(len(my_list)):
    if my_list[item][1] > max_cols:
        max_cols = my_list[item][1]

max_rows += 1
max_cols += 1

#append missing gaps to the original list
for x in range(max_rows):
    for y in range(max_cols):
        if (x, y) not in my_list:
            my_list.append((x, y))

#find the destination node
destination_cooridinates = None
for key, value in whole_mesh_dict.items():
    if destination_ipaddress == value:
        destination_cooridinates = key
        break

source_cooridinates = globals.my_coordinates
my_list = sorted(my_list, key=lambda k: [k[1], k[0]])
my_list = sorted(my_list, key=lambda k: [k[0], k[1]])
dicty = {}
counter = 0
listy = []
```

```
    for i in my_list:
        dicty[counter] = i
        listy.append(counter)
        counter += 1

    #reshape 1D list into a 2D grid list
    x = np.array(listy)
    a = np.reshape(x, (max_rows, max_cols))
    string_list = []

    #Assign “$” sign for destination coordinates, “#” sign if the
node is down, “.” if the node is up
    for i in range(len(a)):
        temp = ""
        for j in range(len(a[i])):
            if (i, j) in original_list and (i, j) ==
destination_cooridinates:
                temp += "$"
            elif (i, j) in original_list:
                temp += "."
            else:
                temp += "#"
        string_list.append(temp)
    goal = "$”

    # Perform Breadth-First Search with source and destination
coordinates
    def bfs(grid, start):
        queue = collections.deque([[start]])
        seen = set([start])
        while queue:
            path = queue.popleft()
            x, y = path[-1]
            if grid[y][x] == goal:
                return path
            for x2, y2 in ((x + 1, y), (x - 1, y), (x, y + 1), (x, y
- 1)):
```

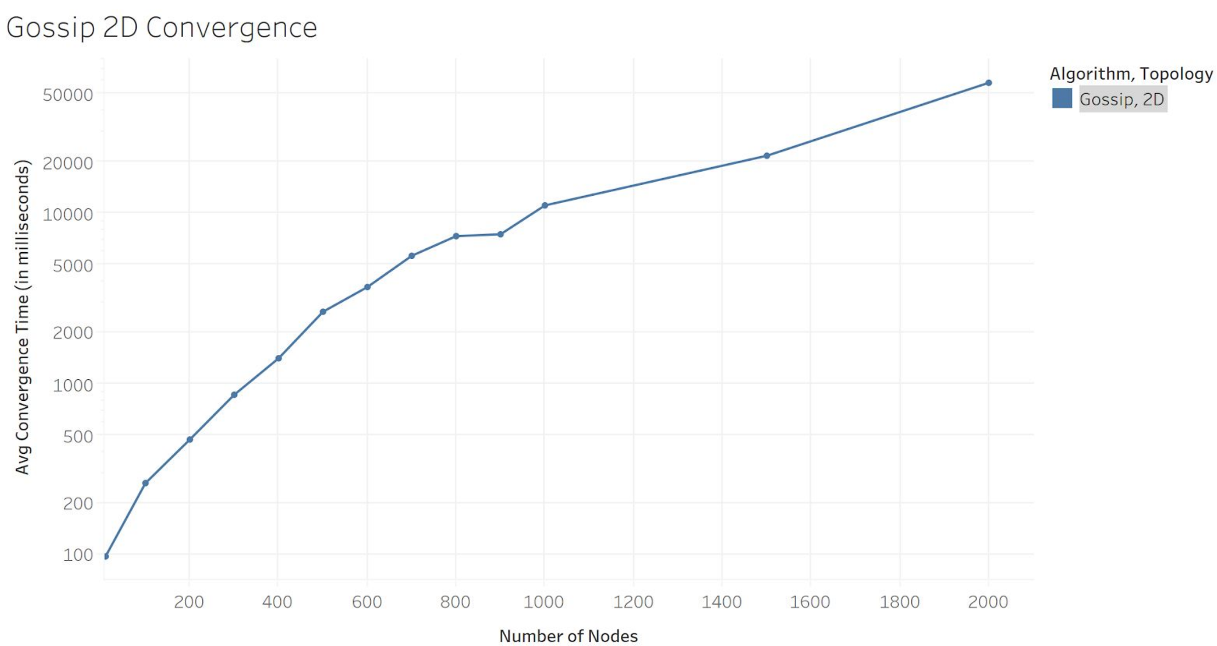
	`` if 0 <= x2 < max_cols and 0 <= y2 < max_rows and grid[y2][x2] != "#" and (x2, y2) not in seen: queue.append(path + [(x2, y2)]) seen.add((x2, y2)) path = bfs(string_list, (source_cooridinates[1], source_cooridinates[0])) new_path = [] for item in path: new_path.append((item[1],item[0]) updated_path = [] for item in new_path: updated_path.append(str(item)) return updated_path ```
Replication Data Forward	``` def replication(path_one, message_stream_of_chunk_bytes, metadata): ''' Establish GRPC connection and forward the data to the neighbor nodes Embed the metadata along with the data to be replicated Call the stub ''' +<CODE to establish gRPC> +<CODE to create metadata and convert the stream of bytes into byte array> request = replication_pb2.FileData(initialReplicaServer=globals.my_ip, bytearray=bytes(message_bytes), vClock=VClock, shortest_path=path_one, currentpos=0) resp = replicate_stub.ReplicateFile(request, metadata = metadata) return resp ```

Timed Results

Time take for Gossip of Gossip to achieve consensus in a 2D network

Convergence criteria:

- 50% of nodes vote on the same information
- Node blacklists itself from gossiping if the same message is received more than 10 times



Retrospective

1. How was your experience integrating with other teams?

There are not many times that we have an opportunity to do a class project which requires every group in class to collaborate. However, it was not a very pleasant experience because we had to choose a topic for ourselves but we did not have any clear communication at the beginning which led to the project started late. Even though we got to learn a lot during the building process, but the inter-team time clashed for meetings and the missing sync between the teams failed everyone. Everyone was so busy fixing their issues till the end, the assumptions and even designs were changed a lot within teams throughout the development resulting in further issues with the dependent teams.

Most of the teams were (except for the Memory Storage, as per our knowledge) in their development phase till half an hour ago on the day of the presentation. Since our team had dependencies on 3 Teams (Fault Tolerance, Mesh, and Memory Storage), our team Based tested code failed because of conflicting assumptions in the code from different teams. This environment made testing very difficult, since dependencies were in constant state of flux till the point of integration.

Even our team cannot claim that our module was ready to be integrated weeks ago but at the team level, we had been constantly working on our module. On December 2nd [The first scheduled integration], The Mesh team made a major change in their architecture by shifting from fully connected to a 2D Mesh. All the code from before had to be reconstructed.

2. What went well and what could have been done better?

During implementation, if the design mesh design was consistent, the result could have been better. However, we had to rewrite the code while still being dependent on other teams, our Gossip and Consistency groups could not make the replication system function properly. We believe that if we receive better communication from others, especially with the Mesh team with 2D Grid creation, we could fully make the data replication work in this distributed system.

3. Would a smaller team-only project provide a better experience? Why or why not?

Yes, because a small team is good enough for cooperation and integration focus.

On the large scale of cooperation, all of the development time will fall into the high level discussion, communication, and integration. Without high level well defined architecture, large team will cause more confusion, and unexpected changes along with development process. The large project requires execution plans which lay out what task is the highest priority. However, in our class, we have too much dependence from one to another team, and this led to unclear communication.

4. Compare your design expectations with the actual implementation.

As briefly mentioned, the original design called for creating a fully connected mesh. If this design had continued to the final implementation, it would have led to faster convergence times and the gossip about gossip would find a replica candidate that is connected to the node. This means that the shortest path to the node does not have to be found, which also means that a logical representation of the 2D mesh does not have to be recreated.

5. Feedback for future classes

For future classes, we believe it could be better if we have smaller team-only project. However, class project could still be manageable if Professor Gash can assign more detailed requirements so that there can be less confusion between groups.

Individual Contributions :

Vaishali Koul :

- Implemented Gossip of Gossip
    - Evaluate gossip information received and gossip the updated information
    - Retrieve neighbor capacities using gRPC calls to network team
- Virtual Consensus
    - Implemented Convergence Criteria
- Integration and Testing

Mudambi Seshadri Srinivas :

- Implemented Gossip of Gossip
    - Implemented Transmit function, fetching alive neighbors
    - Implemented each module as a separate thread
    - Implemented gRPC calls to forward data to neighbor nodes
- Virtual Consensus
    - Implemented Convergence criteria
- Integration and Testing

Roberto Campbell :

- Implemented Logical snapshot creation
- Reshaped the 1D list of network metadata to a complete 2D grid by filling the potholes
- Integration and Testing of modules

Tin Vu :

- Implemented Logical snapshot creation of the 2D grid
- Worked on the Generation of paths by handling failures along with the Consistency team.
- Integration and Testing of modules

Phuong Tran :

- Locally created the vector clocks at the initial write node to be passed on to replicas.
- Worked on the Generation of paths by handling failures along with the Consistency team.
- Integration and Testing of modules

References

[1] Gossip Based Computation of Aggregate Information

[2] The Promise, And Limitations, of Gossip Protocols

[3] Epidemic Algorithms for Replicated Database Maintenance

[4] Gossip Protocols

[5] Implementation of Gossip Protocol Using Elixir

[6] Estimate Aggregates on a Peer-to-Peer Network

[7] Gossip and Epidemic Protocol

[8] A General Explanation of Gossip about Gossip and How It Works

[9] Breadth First Search Tutorials & Notes: Algorithms

[10] Vector Clocks

[11] The SWIRLDS Hashgraph Consensus Algorithm: Fair, Fast, Byzantine Fault Tolerance

[12] Hashgraph the Future of Decentralized Technology and the End of Blockchain