Dataverse Project

Shubham Kumar
Aditya Mantri
Pranav Patil
Manav Rajvanshi

May 18th, 2020

Table of Contents

Introduction **3**

Approach **3**

Technical Emphasis **4**

- Investigation 5
- Moving data closer to the client 5
- Server logs 6
- AWS Deployment 7
- Monitoring Utilization 7

Conclusion & Teamwork **7**

Citation **8**

GitHub: https://github.com/pranavpatilsce/Distributed-Systems-P2P-Network/tree/master

Introduction

This project's intention was to set up an overlay network and be able to collect, store, and stream data back and forth across a distributed network using gRPC and Protobuf. We have built the toolkit level project that can be integrated into another project if they choose to handle their data in this way.

Approach

The approach for this project was to focus on the backend implementation of gRPC and Protobuf in satisfying the main requirements of this project. We used APIs to wrap around the core transport of data via gRPC. Earlier when we started the project, we explored the idea of using Docker swarm because Docker provided the setup for setting up an overlay network. When we looked at the configuration and setup of it, we realized that we can't control the swarm joining process and make a network that is discovery-based. Hence we built a network manually using EC2 instances. To handle the connection part of the network, we have provided a config endpoint via which the user can connect instances together.

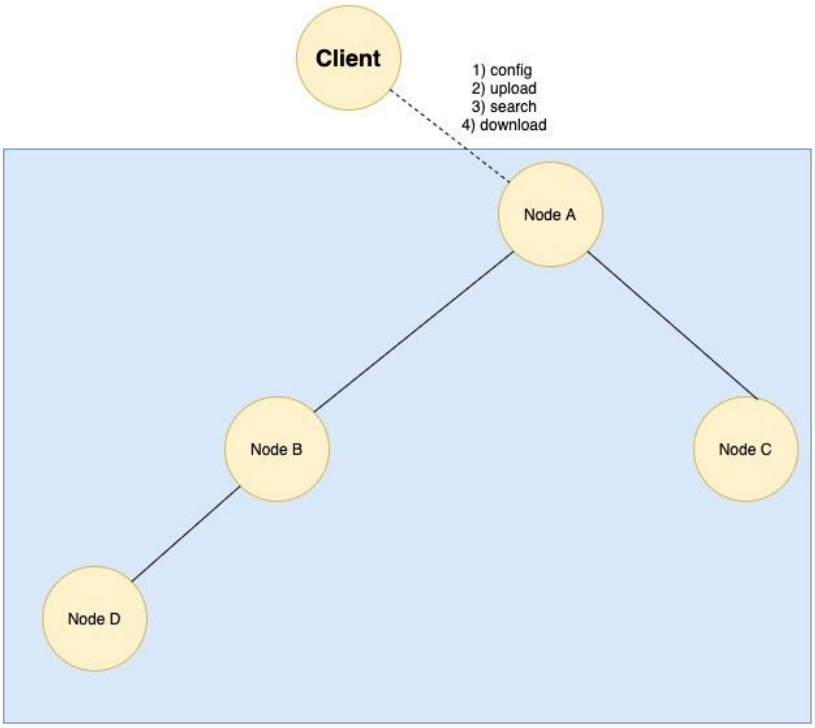

Image #1: Shows a network that we presented in our project demo. The client can connect to any of the nodes. We can have N number of clients connected using '/' endpoint and the clients are connected when they want to upload/search a file. We can also have N number of nodes connected using /config endpoint.

Technical Emphasis

This section will include an explanation and reasoning of the major code files in our project. We will go through the proto file, the client, and the server files. The project is set up so that the client.py is running a frontend interface with RestAPIs that handle uploading files to the network, downloading files from the network, and setting an entry point for the network.

The proto file has a service object with 3 services declared (upload, search, and config). Each of those services are called from the 'client.py' file since the client.py is the file that hosts incoming API requests. Those requests (for example '/' endpoint) take in the file. Once the file is taken in, it calls the gRPC server.py using the dataverse_pb2 that was generated from our proto file. It sends it to dataverse_pb2 using the ImageUploadRequest proto message. All of the endpoints we have in client.py use dataverse_pb2 to send data via gRPC to server.py.

The storage of the file happens on the server.py where the file is received via gRPC to server.py and the logic on that end serves the purpose of keeping track of file requests. To respond back to the client.py with a response, server.py uses dataverse_pb2_grpc to send back the request to the client.py and it shows using the 'render_template_string' on a web frontend. The client can

now download the file. Internally, the /download API is used to handle the download onto the client side the file they searched and requested to be downloaded.

Investigation

Our search is based on **'Dynamic Discovery'** (No fixed list of IPs) of nodes, this essentially means 'Search in a graph of connected nodes'. The two main search algorithms used for graph based search are:

1) Depth First Search
2) Breadth First Search

While choosing the search strategy we figured that both the approaches would be similar in performance for the **FIRST SEARCH** request, however for the subsequent requests BFS would greatly **outperform** the DFS approach. Let's discuss why:

We'll assume that the first search has been done and the data has been moved to the node that client connected to initially (Closest). Now for DFS strategy, if the client connects to a different node on subsequent searches then it may have to search through the entire network (in the worst case) before returning the data to the client. However, the BFS strategy would ensure that we search the nearest nodes next to the current node first and will surely get the data in very few hops as compared to the DFS approach. Hence BFS is better suited for our dynamic discovery model.

Moving data closer to the client

Our system ensures that the data is stored on a node closer to the client requesting the resource (Closeness measured by the number of hops here). Suppose we have our data (a.jpeg) stored on Node D and our client connects to Node A and searches for the file a.jpeg, then Node A would fetch the data from Node D and respond to the client first and then store the data on itself too for any future searches and retrieval of the file.

A requirement for the project was having a feature of data replication in our network. We have implemented a replication factor of 2 where if there is a data on 'Node C' then the data would also show up on 'Node A' (referencing Image#1 above). This is useful if 'Node C' goes down then the data on it is still accessible via other nodes available around it.

Server logs

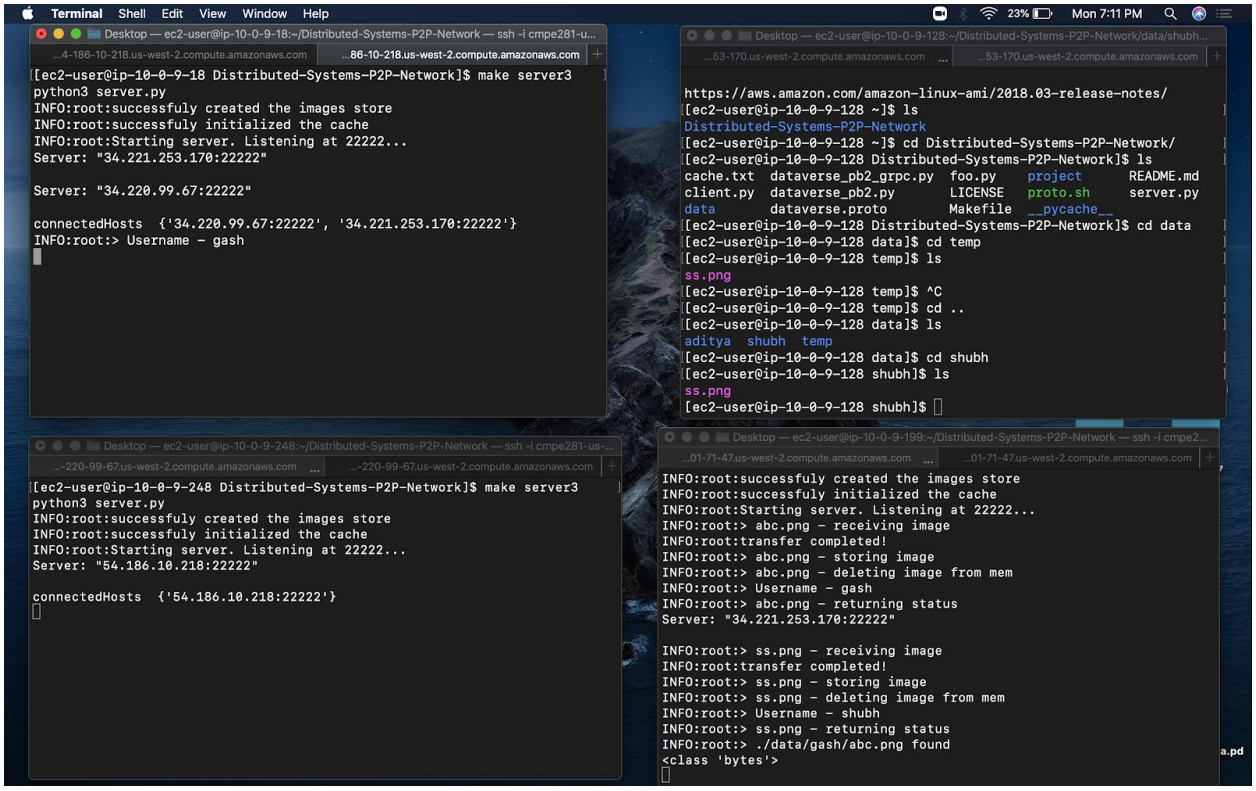

Image #2: As shown in the demo, these are the 4 nodes showing the connection and data transfer between the EC2 instances we set up for the demo.

We have seen through our demo that:

- Relocation: the data is moving closest to the server
- Replication: the data is replicated to other nearby nodes
- Config: the nodes can connect to any other node dynamically
- Search: searching the data file via BFS algorithm
- Upload: a client can connect to any node on the network and upload a file

AWS Deployment

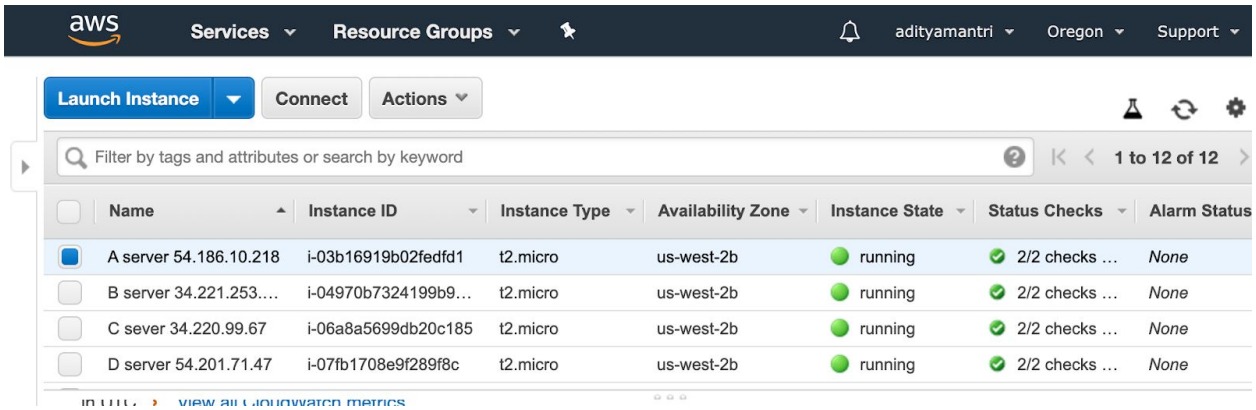

Image #3: Here we have 4 nodes deployed on AWS EC2 instances. These 4 nodes are directly correlated to the Image#1 shown on page #3.

Monitoring Utilization

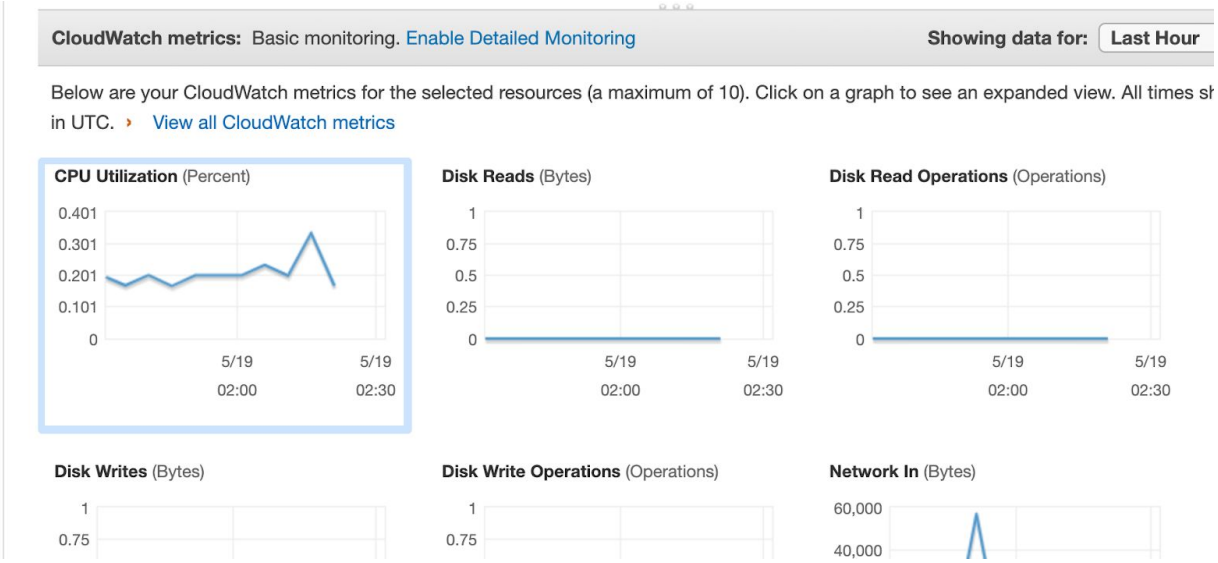

Image #4: This is a monitoring feature that is actually provided by AWS. Here you can see CPU utilization.

Conclusion & Teamwork

There was a lot of trial and error when working on this project. The conversion between python2 and python3 bytes vs unicode string was plaguing us when building the project as sometimes we would write code and not realize how python was processing the data. Apart from that, we

looked at github repositories of other projects to see various approaches and learn from them. Those projects are linked in our citation below. When starting the project, we didn't really know where to start so we started by working on the proto file and ensuring the project can transmit some data. Once that was accomplished, we moved onto transmitting images. Once we understood how the API linked with the gRPC and server.py, we worked on /download and /config endpoints to cover other functionality. Lastly, once we finished coding we tested the setup on 3 separate laptops and checked if uploading the file on 1 laptop was available for download on another laptop. It was a good learning experience and a good way to get our hands dirty with gRPC and learn how the proto file works.

In terms of team work, COVID-19 really forced us to use a distributed way to learn and work together. This was awkward at first and took some time to get used to. Most of the project work was worked on in pairs so we can check each other when making mistakes and run a Google search on the side to look up alternative approaches and official documentation for bugs and errors. Pranav started to investigate how to build the project using Docker swarms. Due to lack of control that we have in swarms and their nature in employing the master/worker node, it didn't seem like a proper overlay network that we could manage. This approach was abandoned and we chose to set up a network manually. Then Shubham and Pranav worked on getting the project started by looking into protobuf and initial gRPC file transfer upload functionality. Manav and Aditya looked at the overlay network side when getting started and read documentation on how to do it manually and cover for other project requirements. Shubham and Aditya then took over the server configuration and network side of the project and Manav and Pranav worked on making search and downloads work for the user. Pair programming was done through sharing screens and verbally saying that code should be and after a feature was finished, it was pushed to git and then pulled by the other programmer to then work on their end when required.

Citation

We were inspired by and used gRPC and protobuf examples widely available on the internet for everyone to see. Below are some links we used to learn about protobuf and gRPC and used elements from these links in our project. Some of the links below are not in Python (we used Python) but they were good examples so we looked at them.

1. https://developers.google.com/protocol-buffers/docs/proto3
2. https://stackoverflow.com/questions/34969446/grpc-image-upload/34982660#34982660
3. https://grpc.io/docs/quickstart/python/
4. https://github.com/grpc/grpc/tree/master/examples
5. https://grpc.io/docs/tutorials/basic/python/
6. https://ops.tips/blog/sending-files-via-grpc/
7. https://github.com/goooooloo/grpc-file-transfer
8. https://github.com/Nirvan101/GRPC-Video-streaming
9. https://github.com/michaelawyu/api_tutorial

10. https://medium.com/google-cloud/building-apis-with-grpc-continued-f53b5a5ab850