CMPE 275
PROJECT 1 REPORT
DISTRIBUTED CHATAPP

To: Prof. Gash
From: Rency Joseph (011429899)
Shefali Munjal(011833562)
Jasmeet Singh (011810058)
Parag Vijayvergia (011818846)

Salient Features :

1. Providing movability/loose coupling to client via efficient inter cluster communication.
2. Implementing Raft for fault tolerance and consensus within the cluster.
3. Client implemented in Python and Servers implementation done in Java.
4. Using thread pool to manage large number of frequently received tasks such and re-establishing channel connection.
5. Persisting Client data in MongoDB.

Overview :

Our cluster is structured as a physical mesh topology with 4 servers. The internal routing detail is configured via a configuration file. Raft has been implemented to provide consensus during data replication and to ensure availability of service in case of node failure.

We are using UDP discovery between client and server as well as from our cluster to discover other clusters in the network. Our system is capable of communicating with other cluster to pull messages for a particular user as well as to serve the similar requests from other clusters.

We are also able to provide movability to clients by being able to handle clients registered on other clusters and full fill their requests.

Our server details are mentioned in a configuration file nodes.conf which is used to startup the system. The file contains all the server host and ports details that is used in the system to prepare connections between the servers.

We have implemented raft as per the paper published -
https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14

Since our system is using an implementation of the RAFT algorithm, we need at least 3 servers to be up and running in-order to select a leader and allowing the

clients to connect to the servers. Our systems is functional as long as majority of the servers are running.
Initially, all of the servers start in the Follower state. **Once at least 3 of the servers from the configuration file are running, an election is issued** by the *server who detects it first* and turns to a candidate state. If it gets majority of votes, it turns to a leader.
The leader then starts a UDP discovery module it's client port to allow the clients to discover it. Once a client is connected to the leader, it performs user registration if the client is entering for the first time.
Also a follower sends a **negative vote** to the candidate **if the lastest log index of the candidate is less than the followers log index**. This ensures that the leader is not outdated. The other scenario where a follower sends a negative vote is when it has already voted for another candidate in the new term. This ensures that a follower votes only once in a term.
Once the user is verified and this is not the first time the user has registered into the system, our system checks if there are any messages for this user on our / neighbouring cluster servers and if yes, **push the unread messages to the user**. The user does not need to explicitly issues a pull request here.
When a message is sent to this user while it is online, again **it does not need to request a pull operation.** Our server automatically pushes messages to them.
All the messages and user registration details are replicated on all servers using the raft system.
The leader first redirects the client to the follower in a round robin fashion, who will then work for the client. Once the client sends a new request (message or user registration), the follower redirects that request to the leader who puts it into its log, and then sends an AppendEntries Packet to each of the follower servers to allow replication. Each of the followers also add it to their own logs and confirms the same to the leader. Once atleast the majority of the servers have confirmed, the leader commits that record on it's DB and also sends a commit to all of the follower servers.Also, if receiver ID channel is open with any of these followers, it send the message directly to the receiver and the follower inform the leader that the record has been read. The leader then replicates this read status on all followers. Heartbeat is used to check if the leader is alive. The **leader after every 500ms sends an AppendEntries** Packet. If there is data to be replicated, it send the data in the Append entry packet. Else the AppendEntries Packet is sent without any data. The **followers consider the AppendEntries packet as a heartbeat.**

The hero of our system is the **ThreadPooling** that uses its inbuilt queue. All incoming and outgoing are processed on a thread pool (different than netty's worker group). This prevents our system to consume too much resources in peak load situations. Although there is a small trade-off with latency.

Channel management: We are using a channel per client connection. Also, among servers, we are using one channel for to and from communication between two servers.

Client registration details and messages are being saved in persistent database across the cluster. This ensure that client information is always available and continuity of server even in the face of node failure. MongoDB is our choice of persistent database because of it ease of storing and retrieving data in readable format. The client is implemented in Python and can send various messages to one of the servers like, UDP Discovery, User Registration and a Message. The client is also able to receive a message from another client via one of the servers and it is capable of decoding the message from a protobuf type to text.

In our cluster implementation, the UDP service always runs on the leader rather than running it on all the clusters. This help us in achieving some control over client connection. To distribute equal number of clients on each server of the cluster, we have implemented a round robin based scheduling technique, such that subsequent client is always assigned to the next server in cluster. This helps us overcome the problem of starvation among the servers.

To deliver the messages to a client when it connects to a server in the cluster, we are using the "push" method. Thus, any message that was received for a client and has not been delivered earlier, are pushed to the client in real time.

On the other hand, the communication between the external cluster and our cluster is based on "pull" method. When our cluster receives a pull message request from an external cluster, we gather all the messages belonging to the intended receiver and send a reply back to the requesting cluster with the message payload.

Request and response messages are defined under a protobuf which is common among all the clusters in the network. It is defined to be able to distinguish between the type of message sent by the client and an external cluster, while both using the same port for communication.

Test cases -

1. *Leader election*
    a. we have 5 nodes in the system configuration. When at most 2 nodes are up, no election is issued as the majority of servers are not up
    b. Once at least 3 servers are up, an election is issued and leader is elected based on votes.

    c. If a leader goes down, another election is issued and new leader is selected.

2. *Data Replication*
    a. Once a client sends a request to the server, this message is replicated on at least majority of servers via current leader.
3. *User Registration*
    a. If a new user comes in, the user details are first registered into the DB.
4. *Connection Distribution (no starvation as leader performs load balancing)-*
    a. Distribute client requests from leader to follower servers in round robin fashion. I.e. the leader assigns a client to follower1 and then follower2 and so on.
    b. If a client gets disconnected from the server it is connected to due to any reason, it can rediscover the leader and the leader gives the client a new server to work with.
5. *Sending messages 10000*-
    a. Client sent 10000 messages in 5sec
    b. Receiver took 30 secs to receive 10000 messages.
6. *Sending 100K messages* -
    a. Client sent messages in around 45secs
    b. Receiver took around 300 secs to receive these 100K messages.
7. Creating and maintaining HashMap of external clusters -
    a. Being able to search and maintain a map of external cluster nodes, one from each clusters.
    b. Replacing and updating the HashMap if the details of a new node is received from an existing cluster.
8. Sending messages from a client connected to our cluster to another client connected to other cluster.
    a. Tested sending 2000 message from a local client to one external client on each of the 5 external clusters.

<u>What we learnt from this project -</u>

1. Netty framework : Java's Non blocking IO library for writing to channels using event based model.
2. Connection distribution across servers.
3. Multithreaded programming in java (Threads, threadpool) : Using threadpool to handle a large number to small tasks versus using threads for small number number of long running tasks.
4. RAFT implementations (small improvement over original raft algorithm that uses RPC calls for communication, we have implemented asynchronous raft using the sockets.)
5. MongoDB internals

Features to be improved -

1. Currently, we are replicating every data on all servers. Due to this, the node with the lowest capacity is the bottleneck. To improve this, we plan to introduce Replication with Sharding which can help us make use of nodes as per their capacity.
2. Batch processing instead of single commits.
3. Our user registration is very basic and needs improvements.
4. Using quorum for multiple read and write to achieve eventual consistency.

CONTRIBUTIONS :

Rency Joseph:

1. Raft Implementation: Leader election
    a. Transition from follower to candidate based on random timer.
    b. Transition from candidate to leader based on votes from followers.
    c. Term Management.
    d. Ensuring UDP discovery starts and stops as leader starts and stops.
2. Cluster configurations: preparing the configuration design for internal servers.
3. Managing the unread and read messages.
4. Inter Server Connection Monitoring: It includes establishing and maintaining server to server channel connections in a mesh topology(using the capability of the channels being bi-directional).
5. Protobuf management for internal communications. (Voting, Heartbeat)

Shefali Munjal:

1. Raft Implementation:
    a. Node state variables persistence in MongoDB
    b. Log-replication among servers using asynchronous AppendEntries packets
    c. Application of logs to the replicated state machine after log replication on majority of the followers.
    d. Heartbeat broadcast from leader to followers using AppendEntries
2. Thread management:
    a. Implemented a multi-threaded server architecture using thread pools (other than netty's boss and worker groups)
    b. Handled all the incoming and outgoing requests on threadpool's internal queue to keep the resource requirements under check
3. Python client
    a. Length prepend encoder and decoder to communicate with netty's channel pipeline (which uses a 4 byte length prepender)
    b. Command Line interface to send messages to other clients

    c. Socket communication with Netty server using Python's socket package

4. Establishing a real-time communication link between two clients connected to different followers, using RAFT's replicated state machine

<u>Parag Vijayvergia :</u>

1. Implemented UDP discovery server and client using protobuf encoder and decoder. Shared with other teams too in order to maintain uniformity.
2. Configured the MongoDB connections and APIs, to be used by the servers to persist user data received at their ends.
3. Contributed in the Inter cluster protobuf file.
4. Contributed in designing and implementing connection and communication with external cluster. Making the system capable of pulling the message from other external clusters and handling the pull requests received from external clusters for a particular user.
5. Programmatically fetching host address and broadcast address instead of manually, to achieve uniformity across platforms.

<u>Jasmeet Singh:</u>

1. Setting up overlay network between intra server and exchanging protobuf packet within nodes. (Internal communication)
2. Implemented user load distribution on all nodes in cluster in round robin fashion. (Effective use of resources and avoid bottleneck for client connecting only to leader)
3. Handle write request(send message) on nodes other than leader node. (Write operation route through leader)
4. Maintaining sticky user connection on server for live message exchange. (Managing user connection in hashmap)
5. Python client implementation to discover server, decode/encode messages and user creation requests.