

CO4219/CO7219 Internet and Cloud Computing

Javed Khan (219041004)

Group – co7219

Code repo - <https://github.com/jav7zaid/whiteboardapp>

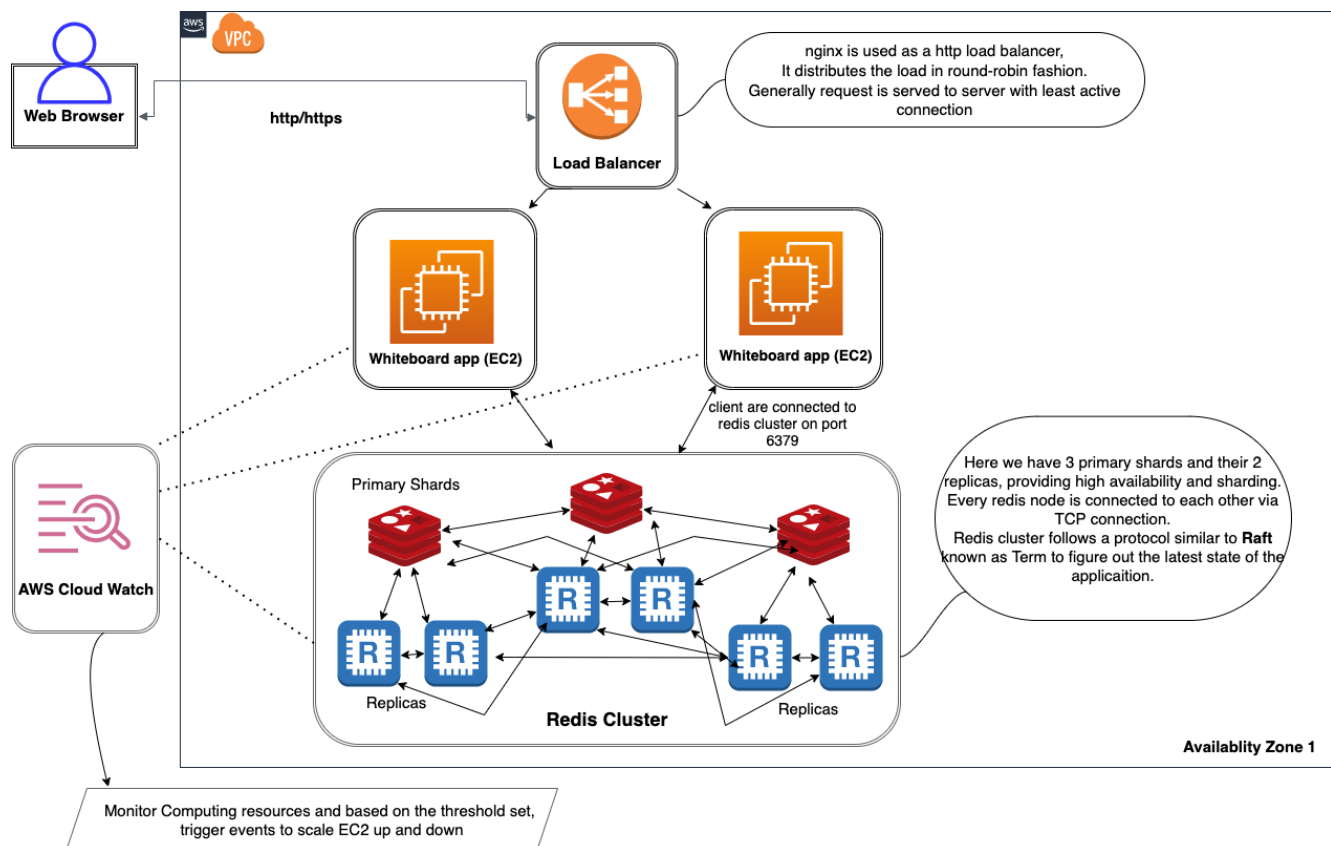
Video URL –

<https://leicester.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=99a14654-cea3-4142-a87e-adf600f41dd1>

References – <https://redis.io/topics/cluster-spec>

<https://redis.io/topics/cluster-tutorial>

Task 1: Produce a Design of your Private Cloud



AWS is a popular cloud services provider, giving the endless ability to the user in terms of creating your own infrastructure, hosting, scaling and pay as per use facility.

I have chosen AWS to implement the private cloud to host my whiteboard application. So, to explain the architecture, AWS provides functionality to implement a Virtual Private Cloud by restricting incoming hosts to policies defined by the user.

In my case I have created security groups that only allow http or https traffic to come in from any IP range. So, a client connected to a browser will be able to access the whiteboard application if the protocol is either http or https, by default AWS serves web application on port 80, I have added a popular reverse proxy nginx to route our request to correct instance of the application running. Basically, nginx distributes the request to server having least active connections or in our case having less load.

The web application is written in HTML, JavaScript and uses express/Node js to serve the application.

I have used AWS EC2 facility to spin up Ubuntu instances, which serve the application back to the client. The EC2 instances are part of Auto scaling group which helps to auto manage the launch of any additional instances based on a trigger policy defined by the user. In our case the policy or auto trigger policy is checking the average Network in >3M bytes. So basically, if any of our instance experience network traffic which sees that in-bytes are greater than the target byte, AWS will launch an additional instance to compensate for that.

This allows our whiteboard to auto scale horizontally to serve more clients and balance the load consistently.

The application uses Redis cluster as the primary database, Redis is a in memory database it provides high availability and sharding.

Here to store the state of the application, I have used Redis Cluster mechanism.

Redis cluster allows us to store the data in shards, we are using configuration that contains 3 Primary shards and each shard is having 2 replicas to better provide availability and prevent any fault that might happen within a system.

Shards are logical units within a DB, it used to share the data across all the nodes within a Cluster. Our model here is an eventual consistent model, as it takes time for the updates to reach the replicas. Every cluster is connected to other via TCP connection, it allows nodes to communicate with each other and in case of any failure of the primary database the replica will take over the role to maintain a consistent system and reduce any failover.

There is a trade-off here is, we are letting go of our consistency, for high availability and fault tolerance.

As said earlier because we are developing a real time application the availability factor is more prevalent in a way that we can work with eventual consistency, as we know initially results are inconsistent but provide much faster with low latency.

The user will be seeing real time drawing of the other users that are connected to the application, using redis allows to provide multiple instances of the application to be available with the latest data eventually coming up to the instance of the application user is connected to.

When a user connects to a new application, a request is sent to the redis cluster to fetch the initial state if any, and when user draws any figure on the whiteboard app, web sockets connect all our different client to serve the application in real time.

This design pattern offers low latency, high availability having a shared topology network configuration, where both the instances communicate with each other via redis DB.

I have also used AWS cloud watch service to monitor the health of EC2 instances and redis cluster, these parameters generally include the average CPU utilization, Network In/Out packets, failed nodes, and DB memory percentage use for redis instances.

Although it seems like a shared memory database but, instead it is a cluster-based system where each and every redis shard and replica has different endpoints from which it serves the request, hence it is more like a distributed system, so every user will have a different primary shard it is talking or connected to.

Task 2: Implement the Private Cloud

AWS offers the ability to host or configure your cloud environment in many ways, my environment configuration is standard, where I have used ubuntu to power my 2 virtual machine or EC2 instances. These EC2 machine are configured with 8 GB of mem and is behind a VPC. Below fig. is configuration for a single EC2 instance of that runs the web app.

Instance summary for i-032ad81f01dbb8a2c (webserver2)

Instance ID: i-032ad81f01dbb8a2c (webserver2)

IPv6 address: -

Hostname type: IP name: ip-172-31-4-99.us-east-2.compute.internal

Instance type: t2.micro

AWS Compute Optimizer finding: Opt-in to AWS Compute Optimizer for recommendations. | [Learn more](#)

Public IPv4 address: 18.119.128.217 | [open address](#)

Instance state: **Running**

Private IP DNS name (IPv4 only): ip-172-31-4-99.us-east-2.compute.internal

Elastic IP addresses: -

IAM Role: -

Private IPv4 addresses: 172.31.4.99

Public IPv4 DNS: ec2-18-119-128-

Answer private resource IPv4 (A): -

VPC ID: vpc-0f5edebeeb4

Subnet ID: subnet-01bf8fdb

Instance details

Platform: Ubuntu (Inferred)

Platform details: Linux/UNIX

Launch time: Mon Dec 06 2021 09:33:21 GMT+0000 (Greenwich Mean Time) (about 3 hours)

Stop-hibernate behavior: disabled

State transition reason: -

AMI ID: ami-0629230e074c580f2

AMI name: ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20211021

AMI location: 099720109477/ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20211021

AMI Launch index: 0

Credit specification: -

Monitoring: detailed

Termination protector: Disabled

Lifecycle: normal

Key pair name: awsapp

Kernel ID: -

Once the EC2 configuration is done we can instantiate it, our EC2 application uses a security group which will only allow http or https traffic coming within an instance from port 80. Once the cluster is setup, we will generate our Redis cluster using AWS Elastic Cache service, the beauty of this service is that no matter how many nodes of redis we have it will allow us to serve it using a single endpoint.

In my application we have implements 3 Primary shard nodes and 2 replicas each of the primary, like below, redis-0001, redis-002 and redis-003 are primary shards with logical spaces divides between them from 0-16383.

These db instances will share the data between them and update the replicas.

Name: redis

Shards | Logs

Add shards | Delete shards | Rebalance Slot distribution | Add replicas | Del

Shard Name	Nodes	Status	Slots/Keyspaces
redis-0001	3 nodes	available	0-5461
redis-0002	3 nodes	available	5462-10922
redis-0003	3 nodes	available	10923-16383

Name: redis-0001

Description | Nodes

Add Node | Actions

Node Name	Status	Port	Endpoint	ARN	Parameter Group
redis-0001-001	available	6379	redis-0001-001.05sasc.0001.use2.cache.amazonaws.com	arn:aws:elasticache:us-east-2:382470316396:cluster:redis-0001-001	in-sync
redis-0001-002	available	6379	redis-0001-002.05sasc.0001.use2.cache.amazonaws.com	arn:aws:elasticache:us-east-2:382470316396:cluster:redis-0001-002	in-sync
redis-0001-003	available	6379	redis-0001-003.05sasc.0001.use2.cache.amazonaws.com	arn:aws:elasticache:us-east-2:382470316396:cluster:redis-0001-003	in-sync

AWS provided with a single cluster endpoint, which can be hit by the server to access the db

Configuration Endpoint:The configuration endpoint of the cluster is `redis-05sasc.clustercfg.use2.cache.amazonaws.com:6379`

The protocol to have consensus among redis is like Raft, it uses a master slave topology to handle eventual consistency relying on replication from master node to slave nodes. As all nodes are interconnected over a TCP, they constantly ping each other and exchange messages. These messages allow the cluster to determine which shards are alive and when most of the shard's report that a given primary shard is not responding, they agree to trigger a failover and promote replica shards to become a new primary.

So now as the EC2 and redis cluster is setup, we will connect our web application to the DB. To deploy we will ssh into the EC2 machine via command prompt and then clone the code from the git repository. We are using nginx to proxy all our incoming request to actual port running our backend server.

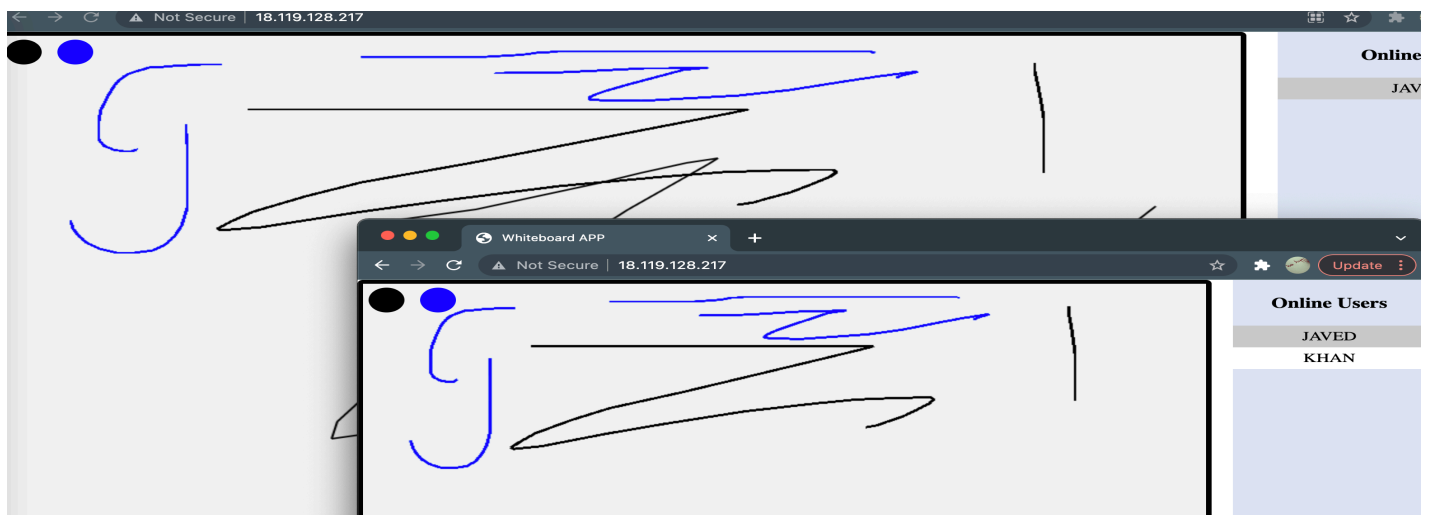
Our server uses Node js and express to serve all redis DB request and expose our client application to port 3000. Below code connects to our AWS redis cluster and open a server UI. We can now access our application on the public IPv4 provided by AWS.

```
const http = require('http').createServer(app).listen(3000, () => console.log('running'));
const io = require('socket.io')(http);
const Redis = require('ioredis');
//const redisClient = new Redis();

app.use(express.static(__dirname + '/ui'));

const users = {};

const cluster = new Redis.Cluster(
  [
    {
      port: 6379, host: 'redis.05sasc.clustercfg.use2.cache.amazonaws.com'
    },
  ],
  {
    slotsRefreshTimeout: 2000,
    dnsLookup: (address, callback) => callback(null, address),
  },
);
```



Task 3: Development of a Distributed whiteboard Application

The development of the whiteboard is done using HTML and Javascript as primary technologies on the Client side. WebSockets library socket.io is used to connect multiple clients to server socket, which listens to events emitted either by server or client and broadcast the figure back to all client connected to that socket.

I have created a server that readily communicates with the Redis Cluster to store the current state of the application using the cluster configuration end point given by AWS, as seen in last task.

```
socket.on('joinroom', key => {
  socket.join(key);
  cluster.get('paint', (err,data) => {
    const history = JSON.parse(data);
    socket.to(key).emit('drawing', history);
  });
});

socket.on('drawing', (data) => {
  cluster.set('paint', JSON.stringify(data));
  socket.broadcast.emit('drawing', data);
});

socket.on('user-name', name => {
  users[socket.id] = name;
  socket.emit('user-connected', users);
});
```

The above code listens to every new connection, or a new user joined, and then query the redis cluster to check if any previous state is present or not, if there is any state or figure previously created by the user it will be broadcasted to every user connected. Every user is connected to new room which is basically a channel on which all users listen to and then broadcast the latest figure to every other clients. When a new user starts drawing a figure the data is then set on the cluster in form of shards in the 3 primary shard nodes and the updates is sent to the slave nodes in order to perform synching.

To maintain consistency throughout we are saving our state in Redis cluster created via AWS Elastic Cache, as our system is eventually consistent and not strong consistent queried results are less consistent early on but has high availability and low latency. Early results of data queries may not have recent updates because it takes time for updates to reach replicas across a DB cluster.

[Code repo - <https://github.com/jav7zaid/whiteboardapp>]

Task 4.1: Demonstrate the Private Cloud

My contribution to design the system was end to end, from building the application to deploying it in AWS.

The web app has two parts namely client and server. Client-side application presents the user with the whiteboard to draw figures, type text etc. Multiple client applications are handled web sockets to allow interactions to the whiteboard in real time. Every user connected to the application is assigned a room or channel to which it is subscribed to, so every new state is then broadcasted to all people within the room. I have used socket.io framework to have real time state replication for every user connected to the web app.



Server-side of the application handles connectivity to the database and creates a web server and responds to any events emitted by the client to the other clients on the connection. I have connected to the redis cluster using ioredis library, this library provides functionality on top of existing redis commands but simpler. Using AWS Elastic cache to create a redis cluster gives us exposed end point to connect to, instead of connecting individual redis nodes using their IP and Port. In our configuration we are using 3 primary shards and 2 replica nodes for each one, this helps us to distribute data linearly in different logical units. It's the responsibility of the master node to update the state to all the slaves or replicas.

AWS setup, I have created 2 ec2 instances that run Ubuntu images with 8 Gig storage. Both instances are part of the same VPC and implement their own security policy to only allow traffic of http or https only to enter the instance. The EC2 are also within the same auto scaling group, which allow us to instantiate another instance of the same type automatically based trigger policy when network bytes entering is >3M bytes,

Execute policy when:

As required to maintain Average Network In at 3000000

To Implement a redis cluster, I have used AWS Elastic Cache service which allows us to setup a cluster of redis instance and expose an endpoint to connect with, cluster config below having 9 nodes in all, 3 Primary shard and 2 replicas each

		redis	Clustered Redis	3	9 nodes	cache.t2.micro	available	up to date	disabled	No	No	-
Name: redis									Global Datastore: -			
Global Datastore Role: -									ARN: arn:aws:elasticache:us-east-2:382470316396:replicationgroup:redis			
Configuration Endpoint: redis.05sasc.clustercfg.use2.cache.amazonaws.com:6379									Status: available			
Primary Endpoint: -									Update Status: up to date			
Engine: Clustered Redis									Reader Endpoint: -			
Engine Version Compatibility: 6.2.5									Node type: cache.t2.micro			
Data tiering: disabled									Availability Zones: us-east-2c			
Shards: 3									Number of Nodes: 9 nodes			
Multi-AZ: disabled									Auto-failover: enabled			
Description: redis									Parameter Group: default.redis6.x.cluster.on (in-sync)			
Subnet Group: redis-sub									Security Group(s): sg-0ee87621af277aecf (VPC) (active), sg-07effc7f6c97dc26a (VPC) (active)			
Notification ARN: Disabled									Maintenance Window: wed:01:30-wed:02:30			
Backup Retention Period: Disabled									Backup Window: Disabled			
Encryption in-transit: No									Redis AUTH Default User Access: No			
Encryption at-rest: No									Customer Managed CMK: -			
User Group: -									User Group association status: -			
Slow log: disabled									Slow log error message: -			
AUTH token last modified date: -									Outpost ARN: -			

Task 5: Critical review of your system

5.1 Architecture choices

I have used AWS to create my design system, AWS is most used cloud platform as it is easy to setup and work on, also it has pre-built functionalities in making it easy to deploy, create instances. Monitor services, all popular DB flavours are available and comes with pre-packages configuration. One of my intentions to select AWS, was because of the ease of use and the proposed design has 2 EC2 instances running the whiteboard application which are then connected to the Redis Cluster to save the state of the application and make it available to the client in real time. My design system support availability and fault -tolerance over consistency. As this a real time application it is given that availability should be takes care first and state will be updated and will be presented to the user later.

As AWS also provided resource watching services like CloudWatch it is easier to monitor resources on different parameter either network, storage, disk utilization etc. Based on these metrics one can then trigger an additional instance automatically.

5.2 Strength and Weakness

Strength –

One of the reasons where my design stands out is the simplicity to work with, as the user can exactly comprehend the functionality of the whiteboard application.

As this a real time application having redis cluster system improves the performance by providing high availability and low latency to the end user.

The cluster-based approach also helps in having any failures as data is shared across multiple shards as well as their replicas.

As there are total 9 nodes serving the state, the write request could then go via any primary shard, although it looks like a shared system it is not instead its more distributed than a traditional system This system provides more replication and raft-based approach to counter failure of any nodes.

Weakness –

Could improve the eventual consistency by implementing a paxos based system.

The load balancer or in my case nginx is the only source for routing, so if that goes down application would be shutdown.

5.3 Suggestions

- Implement a clearer distributed architecture, with each node sitting and running their own DB instance as well.
- Execute raft or paxos based DB to have stronger consistency in the system, but this may lead to higher latency.