Synchronized YouTube video playback application(Group 2)

Apoorva Muthineni amuthine
Chukka Bhargav bhargavc
Tanvi Gupta guptat2

1. Introduction

Social interactions over the internet have become an integral part of our lives. Sophisticated use of internet protocols and high-speed network connections surpassed large geographical distances and provided its users with quick access to a wide range of resources like text files, video, and audio files. This coupled with attractive user interfaces and feature-rich user browsing capabilities accelerated the sharing and wide-scale interactions between the users.

To enable this high interactivity, developers aim at creating applications that provide users with an almost in-person like experience. Such a goal is difficult to realize because despite the network having high carrying capacity there is an upper-bound to the achievable speeds over public internets and thus needs to be optimized in the application level with intelligent design and performance metric usage. Along those principles we designed and created a group YouTube video viewing application. It features high accuracy video content streaming among the group of users aiming at providing a *seamless group-watching experience* over the internet.

2. Related work

The real-time media sharing applications are supported through well-known distributed computing architectures and underlying networking protocols. Synchronized maestro schemes (SMS) proposed in [1] uses a centralized client-server architecture where a server entity monitors the client side multimedia synchronization. Inter Destination Multimedia Synchronization (IDMS) [3] is built upon the existing SMS schemes and provides synchronization support by adding an underlying network protocol known as RTSP (Real Time Streaming protocol) [2]. This enables clients to send out streaming feedback messages which serve as a separate feedback channel from the client allowing the server to gauge the client-side bandwidth and adjust the multimedia transmission rates accordingly. The proposed architecture consists of several key components namely, media server, sync client and the sync manager. Such an architecture separates out the roles of the media content distribution from the media synchronization and allows them to function and interact with clients independent of each other. In our project, we closely follow the design choice of separating out media management and synchronization entities and make use of a client based feedback allowing to keep the clients in sync. However, we eliminate the possibility of the server being a single point of failure by extending this fine-grain control by distributing the server functionality across multiple servers. This serves to provide resilience in the case of application failure and solves the problem of overwhelming a single server with heavy requests. Our solution uses a load balancer

to route requests from the client to the server and utilize a distributed in-memory database [5] to maintain communication and consistency between the multiple servers.

Another commonly used architecture in multimedia synchronization is the peer-to-peer communication where information is disseminated over an overlay network. Applications such as Splitstream [6], built on top of Pastry [7], divides data into “stripes” and disseminates over a forest of multicast trees. The architecture supports synchronization and timing guarantees by balancing the load among participating nodes. In pursuit of a peer-to-peer application deployment, we evaluated the APIs provided by the WebRTC [8] framework which enables distributed communication features for web applications. However, this framework supports one-to-one communication between clients and is not robust enough to support group communication and synchronization.

3. Architecture

3.1. Overview of the architecture:

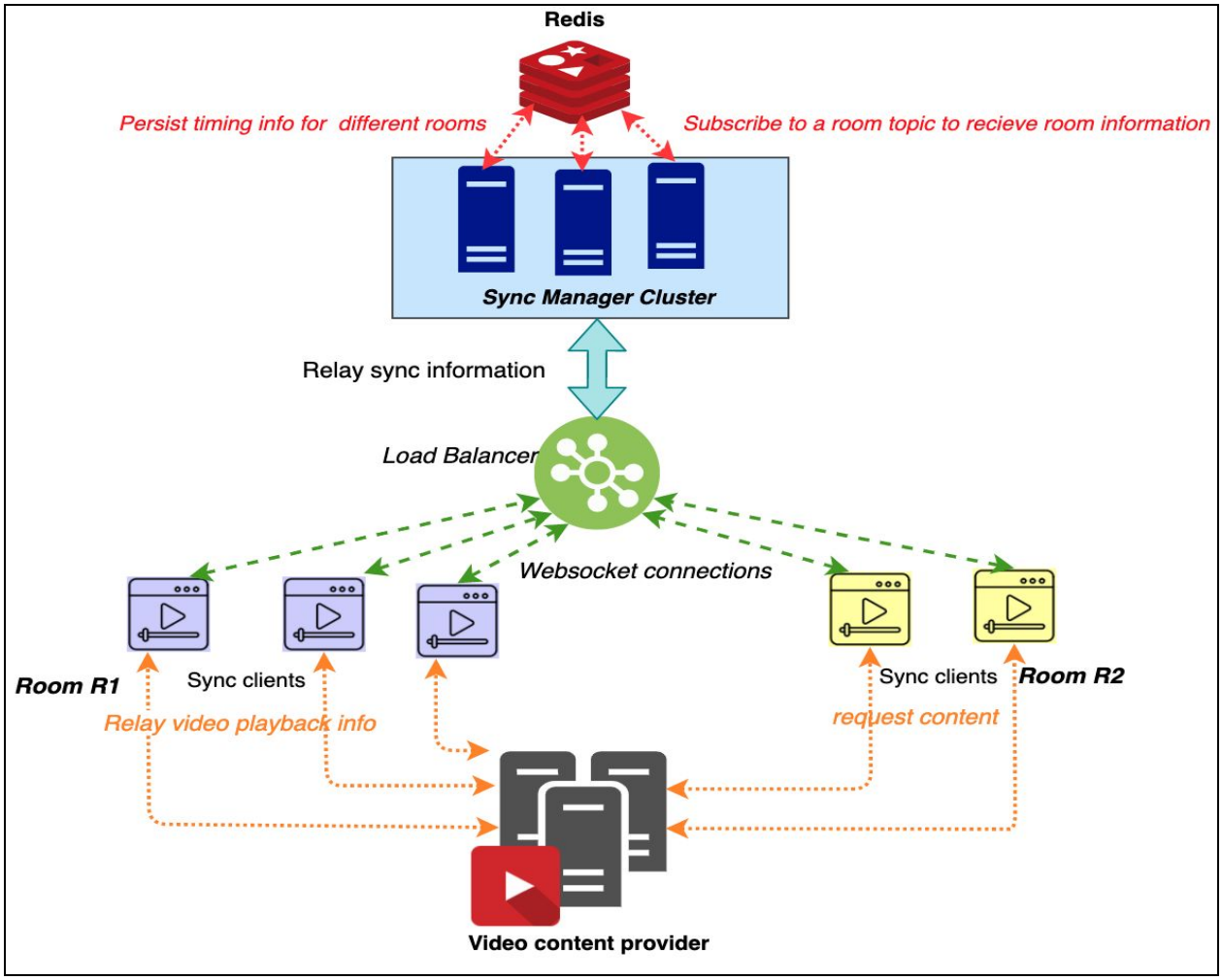

Figure 1: Architecture Diagram

In our architecture, we mainly have three components, Sync Clients, Sync Manager, and Load Balancer. The video content is streamed from the Youtube application. Sync clients are the clients who are interested in consuming the video content in a synchronized fashion. All the clients who are interested in the same content are expected to join the same *room*. Sync Managers are the servers responsible for ensuring the synchronized streaming experience for the clients of each room. We use web sockets[9] to enable two-way communication between sync clients and sync managers. Sync Managers publish the pause/play/buffer events that originate from one of the clients to all the clients belonging to that room. They also receive the current timing positions sent periodically by all the clients of each room and periodically calculate the ideal timing position for all the clients. Additionally, the load balancer is used to route the web socket requests from clients to the least loaded server.

3.1. Sync clients:

Sync clients represent the client side of the application. They have the ability to either create a new room or join an existing room. All the clients belonging to a particular room have control over the video being played. The Youtube application is integrated into the GUI of our application using an iframe. This way, clients have the controls like pause/play/forward/backward/change videos which are available in the typical Youtube application.

When clients create/join a room, they first establish a web socket connection with one of the Sync Manager servers via a load balancer. This connection remains persistent throughout the client's session. At the server end, a web socket topic is created per room to which all the clients of that room subscribe. Whenever a client pauses, plays, or uses any other video controls, it sends a message to the server using the pre-established WebSocket connection. The server then publishes the event to other clients of that room which in turn mimic the controls at their end to stay in sync with the other clients. Additionally, clients also send their individual video timing positions to the server periodically. They also receive the periodical ideal timing positions sent by the server and adjust their playback position to the received ideal time plus the estimated time lag between the client and server.

3.2. Load Balancer:

We use HAProxy[4], an open-source high availability load balancer to route the web socket connection requests from the clients to the appropriate servers. It is configured to route the new requests to the server with the least connections. Unlike HTTP connections which are stateless, web socket connections remain persistent throughout the session. That means, once the client establishes a web socket connection with a particular server(via load balancer) initially, it always sends and receives further messages from the same server whereas the HTTP requests from the clients can be routed to any of the available servers.

Moreover, the load balancer is mainly used to make the application *scalable* and *fault-tolerant*. Whenever the application load increases, new sync manager servers can be

seamlessly integrated into the cluster with a simple configuration file change in the HAproxy. Moreover, in case of server failures, all the clients connected to that server re-establish web socket connections with any of the available servers through the load balancer. Thus, the load balancer facilitates scaling and gracefully handling the server failures in the application.

3.3. Sync Manager:

As introduced earlier, the sync manager serves the sync client requests to create or join a room, periodically relays the ideal timing position to all clients through a WebSocket connection, forwards the sync client events like pause/buffer/play videos to the other sync clients in a room. Considering the scalability and reliability of the system, the sync manager's responsibility is distributed to a cluster of servers.

Through the load balancer, the sync clients belonging to a room can establish a WebSocket connection with any of the sync manager servers. From the use cases mentioned before, we can infer that the servers should communicate among themselves for forwarding information to all sync clients in a room. Moreover, we need a storage system to store all room information and status, sync client's timing updates.

With distributed servers in place, it is a good practice to have a distributed storage system to avoid a single point of failure. In order to achieve highly synchronized playout, the servers should be able to compute the ideal timing position of each room and publish it to the sync clients in frequent intervals. For that, we need a storage system that guarantees low latency query times. Also, our application doesn't have high memory requirements and the information stored for inactive rooms can be purged after a timeout. Redis [5] is an appropriate choice for the required specifications. Since it stores most of the data in memory, query latency is low. Redis also provides a publish-subscribe based messaging feature which is used as a communication channel among the sync manager servers. Through this channel, the sync servers relay user action events to other servers.

Each server is responsible to update the ideal timing position of the room to connected(WebSocket) sync clients. A scheduler runs at each server to update the room timing positions to these sync clients. In this process, the server gets the room information of all the sync clients it is responsible for, from Redis. The server checks whether each room's ideal timing position has been computed in the last *'x'*(configurable) minutes. If not updated, the server takes the responsibility of computing the ideal timing position for the room. Each server maintains a set of rooms that it is responsible to compute timing position. The ideal timing for these rooms is periodically computed by the server and persisted to Redis. This enables other servers to get the latest ideal timing position for the room and also ensures continuity for the newly joined clients of the room. This procedure also ensures that the task of computing the ideal timing position for all rooms is distributed among the servers.

3.3.1. Algorithm for Ideal Timing Position:

Sync clients in a room would periodically send their current position (timing event) and sync managers periodically send computed ideal timing positions to their clients. Each client sends

the timing position of the playout and the recorded time when this timing position was sampled. This information is persisted to Redis as room client information. The scheduled job at the server, responsible for computing the ideal timing of a room, queries the room client information stored in Redis. The server then estimates the current position of each room client. The average of the current position of all the clients of a room is computed as the ideal timing position at the time of computation. This ideal timing position and time of computation is persisted to redis as room's information. The sync servers query the room information from Redis to forward it to sync clients.

A pause or break in the video playout at the client for some duration introduces faulty estimation of the client current position by the sync server. This is because the computed current position of playout also includes the paused duration. To overcome this scenario, the server sends the ideal timing position only when the video is being played. The sync clients also send the timing events on receiving a play event. It is to be noted that the server only considers the sync clients whose position sample time is greater than the latest play event time, for computing ideal timing position.

Our algorithm is tightly coupled with the physical clock synchronization between sync clients and sync servers which is achieved through ntp protocol[10]. Here we are not using the physical timestamp to determine the causal dependency of events. Therefore, the few milliseconds of physical timestamp differences between the various endpoints would not affect our algorithm.

3.4. Scenario : Client sends a request to join a room:

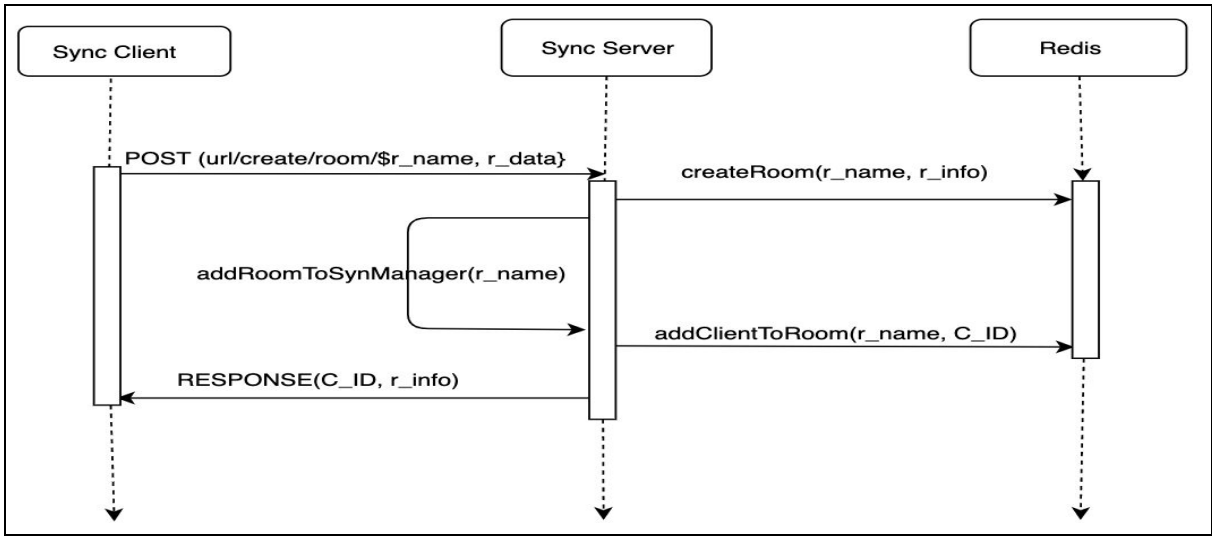

Figure 2: Sequence diagram depicting the series of events that take place when a client creates a room

- A client interacts with the GUI and enters a room name which then generates an HTTP post request containing the room name (r_name).
- The load balancer relays the client request to a particular Sync Manager server which then persists the created room information on the Redis cluster.

(Note: The design follows a DAO(data acquisition object) architecture where a *RoomDAO* provides high level APIs to interface with Redis abstracting away the details from Redis)

- A *Room* object is associated with each room in which the following information is maintained:
    - Video url for the current video being played in the room
    - Video status indicates if the video has been paused or it is currently playing
    - Video position indicates the latest received position of the video from the client event updates.
    - A timestamp to record when the last video status update was received or relayed.
    - A timestamp to record the latest time at which the ideal video position was calculated and sent to the client.
- When a room is created, the *Room* object is initialized to default values.
- Servers share the responsibility of servicing the rooms and they keep track of the rooms they are currently servicing by maintaining their own current room lists.
- The server then creates a unique client ID to identify the client in a room and passes this along with the room name to persist to the associated room Hash Set in Redis.
- Subsequently, the server sends a RESPONSE message to the HTTP post request of the client with the server-created client ID and the Room object containing default configurations as the payload.

4. Evaluation

In this section, we present the experimental setup for each of the components from the architecture described above. For the sake of simplicity, we ran our experiments with two Sync Manager servers as a cluster and accessed the application from multiple clients.

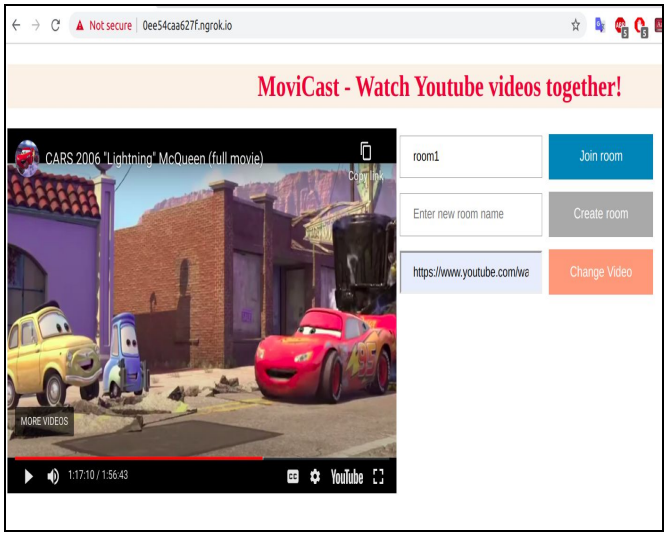

Figure 3: A screenshot of GUI of the application

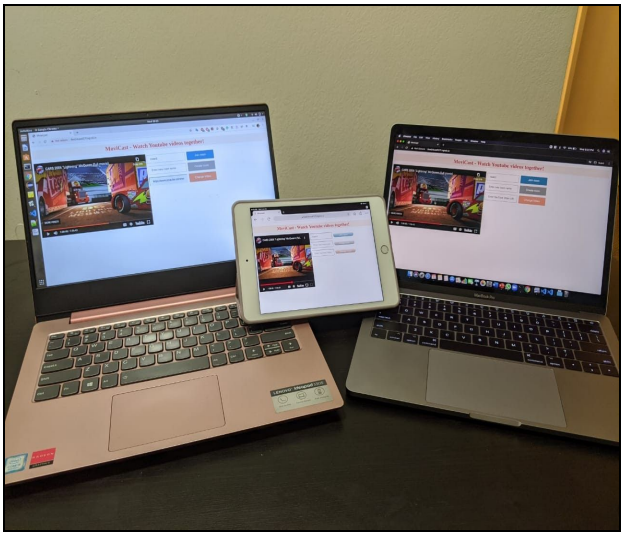

Figure 4: A picture depicting the working application accessed from multiple devices

Figure 3 is a screenshot of the GUI of the application. As explained in the previous sections, users have the option to create/join a room, change video and can also access pause/play/forward/backward controls on the Youtube video iframe. Figure 4 demonstrates the working application accessed from multiple devices where all three devices are connected to a common room. This scenario can be seen as three sync clients accessing the application from different places using the internet. We can see that all three devices are able to stream the same video, perfectly in synchronisation.

Figure 5 is a snapshot of the console logs from the GUI. We can see the Web socket message exchanging between the client and the server. Particularly, we see clients sending the play event to the server which inturn publishes the message to all the clients of that room. Also, the client is periodically sending its current video position to the server and also receiving the ideal video position sent by the sync manager server.

Figure 5: A screenshot of the UI console log showing the websocket message exchange

The below Figure 6 shows the UI dashboard of the HAProxy load balancer. The two backend servers are denoted as ‘srv1’ and ‘srv2’ having 2 and 3 connections each. The green background of the servers indicate the health status of the servers as up and running.

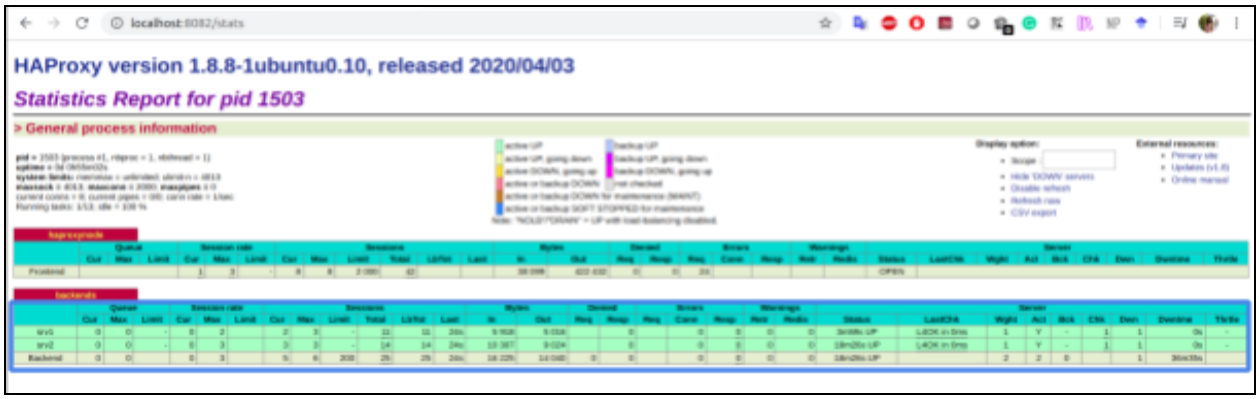

Figure 6: A screenshot of the HAProxy stats dashboard

4.1. Redis Statistics:

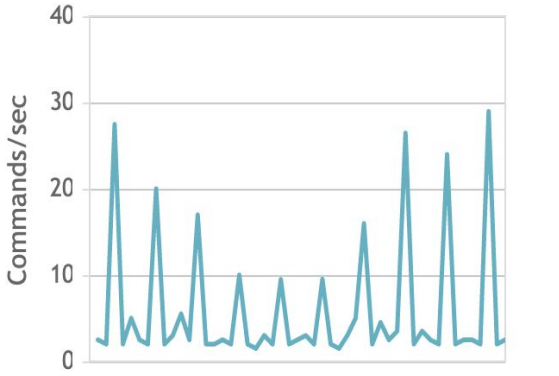

Figure 7: Redis Commands per Second Statistics

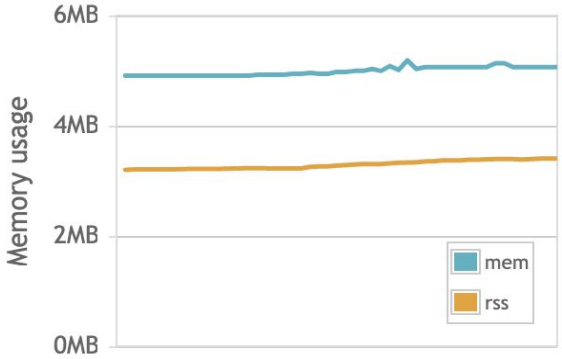

Figure 8: Redis Memory usage Statistics

Figure 7 depicts the frequency of commands per second on the Redis cluster. The spikes are because of the schedule job that is triggered at Sync Server. We can see a sequence of low spikes and high spikes in the graph. This is observed because when the rooms are inactive(video paused or video ended), the Sync Server will not compute the ideal timing position. Otherwise when the rooms are active, to compute the ideal timing position, the Sync Server queries video position information of all the sync clients. So when rooms are active the number of commands to Redis cluster are high.

In Figure 8 the blue plot represents the memory allocated by Redis. The increase in memory usage is because of creating 6 new sync clients and 3 new rooms. The average memory usage for each sync client is significantly low. Since we are only storing current status of room clients and rooms, the memory usage increases only when new clients or rooms are created. This observation supports that our application is not a memory intensive application and strengthens the design choice of using Redis as the storage system.

5. Conclusion

In this project, we aim at designing and creating a seamless video viewing experience among users connected over the internet by implementing fine-grained synchronization among the viewing clients. We achieved this by carrying out a thorough analysis of the available solutions and supported architectures which would best suit the features of our application. We found that fine grained control was best achieved through a centralized architecture and we built our application with a Sync Manager communicating synchronization parameters across all clients and rooms. To address the bottleneck resulting from a centralized architecture, we create a cluster of servers which service the client requests and augment the distribution by using a load balancer. The synchronization calculation is expansive taking into consideration the presence of any slow clients and moderating the synchronization rate among multiple servers by keeping an account of the latest update times.

6. Future Work

We plan to further offload the responsibilities of the sync manager component to a separate component similar to a microservices architecture. In terms of applicability, we would like to enrich the application with a group chat and group audio/video calling feature by applying similar synchronization mechanisms. Our design is robust and we would like to include similar synchronization mechanisms to support other popularly used streaming websites like Netflix, Prime Video, Hulu, etc by enabling authentication.

References

1. Ishibashi, Y., Tasaka, S.: A group synchronization mechanism for live media in multicast communications. IEEE GLOBECOM'97, pp. 746–752 (1997)
2. Inter-Destination Media Synchronization (IDMS) Using the RTP Control Protocol (RTCP) https://tools.ietf.org/html/rfc7272
3. Boronat F., Mekuria R., Montagud M., Cesar P. (2013) Distributed Media Synchronisation for Shared Video Watching: Issues, Challenges and Examples. In: Ramzan N., van Zwol R., Lee JS., Clüver K., Hua XS. (eds) Social Media Retrieval. Computer Communications and Networks. Springer, London
4. "HAProxy Homepage" [Online].Available: http://www.haproxy.org/
5. "Redis Homepage" [Online].Available: https://redis.io/
6. M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth content distribution in cooperative environments. In Proceedings of IPTPS03, Berkeley, USA, Feb. 2003
7. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In Proc. IFIP/ACM Middleware 2001, Heidelberg, Germany, Nov. 2001.
8. "Web Real Time Communication" [Online].Available: https://webrtc.org/
9. "Web Socket Protocol"[Online]. Available: https://tools.ietf.org/html/rfc6455
10. "Network Time Protocol"[Online]. Available: http://www.ntp.org/