vChamber - Synchronised Video Playback

Sizhe Sun
University Of Bristol
wu18332@bristol.ac.uk
Candidate Number: 97198

Jeonghyun Kim
University Of Bristol
cc18316@bristol.ac.uk
Candidate Number: 97485

Weijia Lu
University Of Bristol
tt18284@bristol.ac.uk
Candidate Number: 97564

***Abstract*—vChamber is an online application developed on Oracle Cloud Infrastructure to enable multiple users to stream the same video in a synchronous manner. Users can create a room and invite other users to join, in which video playback progress for are users is synchronised in real time. The application can be run online at `http://vchamber.me` .**

***Index Terms*—Cloud Computing, Client-server systems, Kubernetes**

I. Introduction

A. Motivation

With the development of network infrastructure and media consuming devices, online video sharing and streaming services, e.g. Youtube, has grown significantly over the last decade. The rise of social network enables people to share the video they are watching and their thoughts. However, sometimes sharing a video link via e.g. social media is not enough if you need immediate feedback from your peer. Real time synchronised video playback can bring a different experience, as if viewers are in the same physical room. Coupling with video/voice chat, it can allow immediate feedback to the video content from users. Apart from media consumption use cases, this application can be also useful in scenarios where users want to focus on the same video content, and constantly perform seeking for comments/analysis.

B. Past Work

We found two existing projects that aim to achieve similar objectives to our application: synchronising videos. We researched on both applications and designed our system to maintain their merits and address some issues with them.

Syncplay [1] is an open-source client-server architecture system, which includes a desktop client and a server, written in Python, that aims to synchronises video playback. It uses a custom TCP-based synchronisation protocol to synchronise between clients and the server. The Syncplay desktop client includes a protocol client talking to the server and a VLC plugin controlling the VLC media player. After trying out Syncplay we found it to be generally performing well in terms of synchronising video progress, though occasional bug exists. We found that the requirement of downloading a desktop client and VLC might limit its potential use cases, especially on mobile devices. The syncplay project is still being actively maintained, and hosts a website and a official syncplay protocol server.

Coplay [2] is an open-source web application that supports two users to synchronously playback an online video and video conference at the same time. It uses a very simple synchronisation protocol based on PeerJS, which is a Javascript peer to peer network library based on WebRTC. There is a server written in Javascript to negotiate sessions. The client side is also written in Javascript, deployed as a browser extension that injects into the video playback page to provide video synchronisation on several popular video streaming websites. Similar to syncplay, coplay is not compatible with mobile devices. We were impressed by its intuitive interface and ease to use, but also found it unusable on many websites, mostly due to the video website page design/API changes.

We find that there is no satisfying existing implementation with cross-platform support and neither of the previous implementations has concerned with scalability. We think that this application can be a good example for a cloud application due to its real-time nature and potential scalability challenge under high concurrency.

II. Functionality and Specification

In this section we present the implemented end user functionalities. Due to the time constraint, the scope of this project is limited to implementing the core functionality: synchronising video playback. In order to test and demonstrate the synchronisation functionality, we built a simple static web front end which allows users to create and join a room, in which all users' video playback state is synchronised with some back end server, which is reflected on the front end media player.

A. Rooms

A room constrains the group of users who can playback a synchronised video. A room is created by clicking a button on the front end webpage, which then redirects this user ("the room creator") to the video playback page, also known as in the "room".

In this page the room creator should see a video player and some links that allow the room creator to invite his/her friends to this room. There is some options to allow the user to change the video source. To simplify the protocol and back-end architecture design, we only support Youtube videos.

B. Users

A room consists of two kinds of users. *Master* users are allowed to change the current room-global video playback state, e.g. pausing, resuming, seeking, changing playback rate, etc. *Guest* users might also receive the room state update from the server might cannot change it. For simplicity, each room has exactly one password for master users and one password for guest users. The password is generated upon room creation and cannot be changed. The rationale is that rooms should be easy to create and short-lived, to avoid the hassle of authentication and password management for both the server implementers and the end users.

Due to the time constraint, further features, such as displaying other users in the room, watching videos from sources other than Youtube, etc, are not implemented. However, the basic protocol and this prototype implementation can be further extended to support more features.

III. Implementation

In this section we introduce the design of our systems, discuss some design choices we made, as well as some notable implementation details.

A. Synchronisation and Latency Estimation

Synchronisation is the core function of our application, thus most of our design choices are made around this feature.

For each room, the aim is to synchronise the playback state of all users in the room to a close to consistent state in real time. To decide what we think this state is we define a "authoritative" state of the room that all users in the room should follow. This state is stored at the server, which is broadcast periodically. To change the playback state, a master client sends a message updating the state of the room. When multiple master clients have conflicting state changes, the server gets to decide the new state.

It takes each message some time to arrive from one side to the other, in other ways, when a client sees a message that tell it to resume a video, it is already too late. We take the latency into account by assuming that the latency eventually stablises at some value, and we assume when the one point (server or client) sends a message, the latency for the other side to receive it is close to the stable latency. For example, when we see a message to continue playing at time t_0, and we assume the message took RTT_{est} to get to us, we shall seek to $t_0 + RTT_{\text{est}}$ and resume the playback. It could be argued that the player who initiates such resume operations can get a lead, whereas all other users are potentially missing content from the video. An alternative approach is to make the seeking client wait for other clients to get the message and catch up. We decided to take the first approach since: first, the RTT is usually less than a second so the gap is not that significant; second, the later approach requires the server to estimate the maximum client-to-server latency, which is not easily available without some of our design.

As the clients and the server are connected via network, most likely packet switched network, it takes time for packet to arrive, and the latency can vary due to different conditions, such primarily due to queueing delay. We employ an algorithm presented by Kim and Noble [3] to estimate the latency based on sampling actual ping/pong message round-trip time measurements.

Unlike most many applications where ping messages are initiated by the server, here we decide to let the clients themselves to initiate pings and measure local latencies. This reduces workload and amount of mutable state at the server side. Common problems, such as client lying to the server, is not as important due to the characteristic of our problem.

To tackle asynchronisation caused by network problems and possibly clock drifts, we choose the broadcast the current state message to all clients in the room every few seconds.The rate of broadcast should be set carefully, as a too large period can render the broadcast less useful, and a too small period can flood the server itself.

B. Underlying Protocol

For the vChamber synchronisation protocol, we chose WebSocket as the underlying communication primitive. The primary reason is to reduce the communication latency and to allow the server to initiate a message broadcast (for periodic playback state update). HTTP based protocols are not suitable for this task as they do not maintain a persistent connection, thus unable to give stable latency readings and preserve

An interesting alternative is WebRTC. WebRTC is a peer-to-peer communication protocol, using SCTP, which supports both TCP like behaviour such as ordering guarantees, as well as UDP like behaviour. In theory this would be a more suitable choice for our task, especially under poor network condition when a TCP based protocol e.g. WebSocket would require many re-transmissions, causing significant performance degradation and invalidating our latency assumptions. Unfortunately we decided to not use it for our application due to the lack of complete protocol implementation in modern browsers, in particular the `DataChannel`, especially on mobile platform browsers.

C. Choice of Tools, Platforms and Programming Languages

In order to achieve good concurrency performance, we also look at the choice of programming languages. The server side, deployed on the cloud, is written in Go, and the front end client is written in HTML and Javascript. The primary reason for using Go in back end development is Go's abundant and well-written network libraries. We hope to achieve relatively high concurrency before scaling to multiple instances to maximise our overall performance. For better cross-platform support within the limited development time, we opt for the Javascript media player library *Plyr* [4], due to its awesome cross platform support (seemingly) friendly API. In later development we encountered some issues with Plyr, in particular with matching the media player state with the server modelled state, and separating user-triggered events, vchamber protocol triggered events and events due to e.g. poor network and different behaviours between different underlying media player backends e.g. Youtube and HTML5.

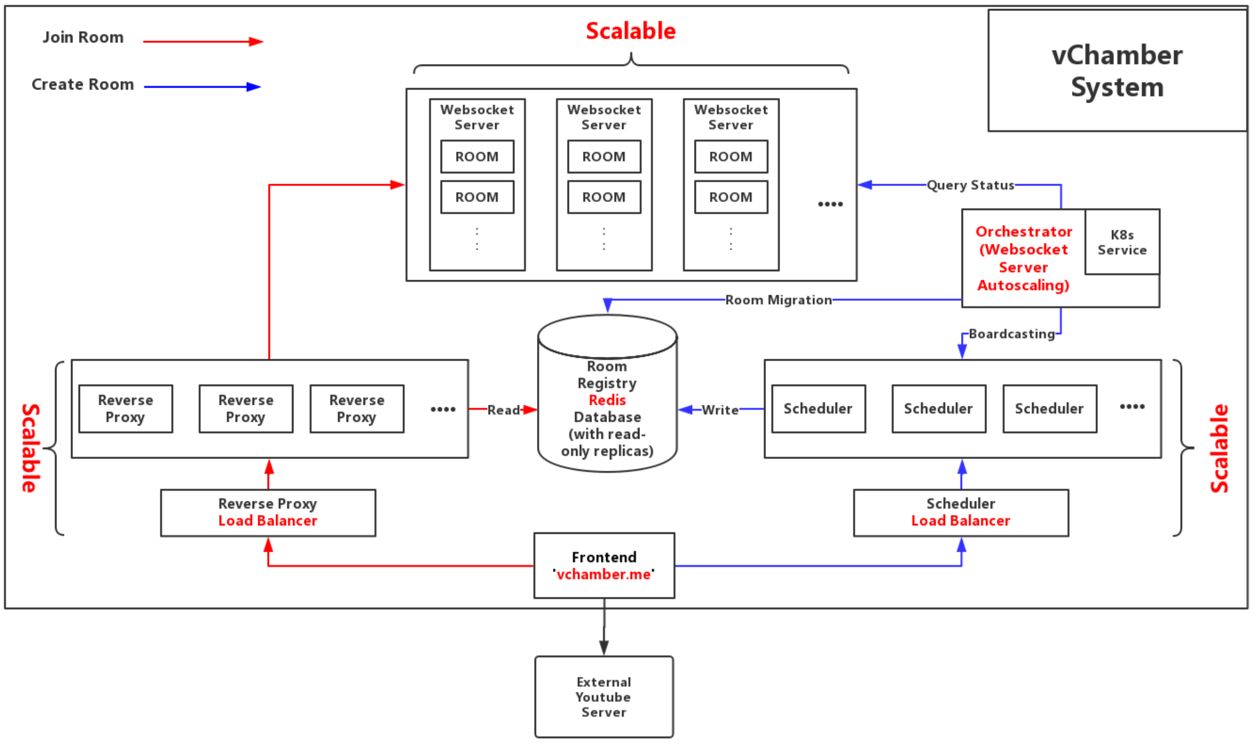

Fig. 1. A diagram illustrating the architecture design.

D. *Cloud Service*

The system is deployed on Oracle Cloud Infrastructure. On the cloud service markets there are many competitors such as Google Cloud and Amazon AWS. These competitors are generally offering more options on computing resources and better documentation. The primary reason for choosing Oracle Cloud is the generous sponsorship from Oracle, which allows us to try out different types of computing resources and infrastructures.

We have chosen to use the Kubernetes Container Engine, which provisions a Kubernetes cluster with three virtual machine nodes as well as the virtual networks required. As shown in the next section, our system includes several communicating components, and it turns out that using Kubernetes managed Docker containers is the easiest way to manage and deploy the whole system as we develop, test and scale different components.

E. *System Architecture and Kubernetes Cluster*

The most performance critical part of the system is the WebSocket based synchronisation server. Considering the potential use cases and protocol complexity, we decided to constrain one room to one single server instance, which means we cannot rely on horizontal scaling to scale up the maximum number of people in one room. This in turn, allows the synchronisation protocol and the server instance design to be relatively simple, which makes our Go server implementation deliver reasonable performance at high concurrency.

In the specification we define that each back end server should expose a WebSocket interface for the synchronisation protocol, and a RESTful API for less time-critical tasks e.g. server/room management. In the scalable design we expose the same set of interfaces to the front end, decoupling the front and back end, and also enables possibly meaningful performance benchmarks.

The focus of the system architecture design is to enable horizontal scaling for high concurrent rooms with relatively smaller number of users per room. The principal is to separate different components, and make each component horizontally scalable.

The system is composed of a static web front end which serves as a portal to the back end servers. The front end Javascript client will also access the video resources from outside our system, i.e. Youtube CDNs. The front end component is a number of pods running Nginx container instances, which are under a load balancer.

The array of back end server pods are arranged as a Kubernetes StatefulSet, as we are binding rooms to specific pods in the stateful set. All other components of the system are somewhat stateless, or are able to recover state from other living components, thus configured as Kubernetes Deployments.

To enable back end scaling, i.e. distributing rooms into different server instances, we need to always know where each room is. We use a room registry based on a Redis in-memory database with read-only replicas and Redis sentinels for automatic failover. Since the room registry information is completely recoverable from the server themselves and generally quickly expires over time, we can make the databases completely volatile, thus stateless and horizontally scalable.

We deploy a series of load-balanced reverse proxies to relay WebSocket connections between clients and back end servers to enable a client to join a pre-created room. These reverse proxies read from the room registry to find the corresponding back end server given the room ID, i.e. they expose the WebSocket interface of a standalone back end server. Since reverse proxies themselves do not write to the database, they can take advantage of the multiple read-only database replicas, thus reducing the connection initialisation latency. Due to the additional layer of data relay, message latency is inevitably increased, yet we have gained better horizontal scalability and protection for the back ends.

We choose to separate the room creation from the room admission part, since room admission requires write (append) to the room registry. Separating the read and write for the database allows more room for performance optimisation in reads. Schedulers are responsible for scheduling new rooms to back end servers based on some scheduling strategy. Thus they expose the RESTful API part of a standalone back end server. Similar to the WebSocket reverse proxies, we choose to trade some latency for better horizontal scalability.

The orchestrator is responsible for monitoring the whole system, publishing scheduling strategy, and decide and coordinate automatic scaling of the WebSocket back end servers. In current design we only allow at most one orchestrator in the system to simplify the orchestrator itself. To achieve fault tolerance for the orchestrator we will need to replicate it and use some distributed system voting algorithm to elect a master and let it make decisions. Since rooms are bound to specific back end servers, scaling down requires cooperation from the server to be scaled down. Due to time constraint, only the observing and policy making part are partially implemented.

IV. Evaluation

To evaluate the performance of our system, we use *Artillery* [5] to load test our HTTP RESTful API and WebSocket interfaces. We also wrote a simple stress testing scripts to perform preliminary tests and help find concurrency bugs.

The figure and the table above shows some performance figures we captured. The room creation test shows that our room horizontal scaling strategy has worked as expected. In the WebSocket connection test, we look at the median, 95th percentile and 99th percentile latency under different loads. At each stage 5000 more virtual clients connects to the server and repeatedly sends Ping message, emulating the most dominant load in our protocol design. With the tolerance value of 100ms latency, our system has survived around 10000 concurrent clients in one room with the 1 ping per second strategy,

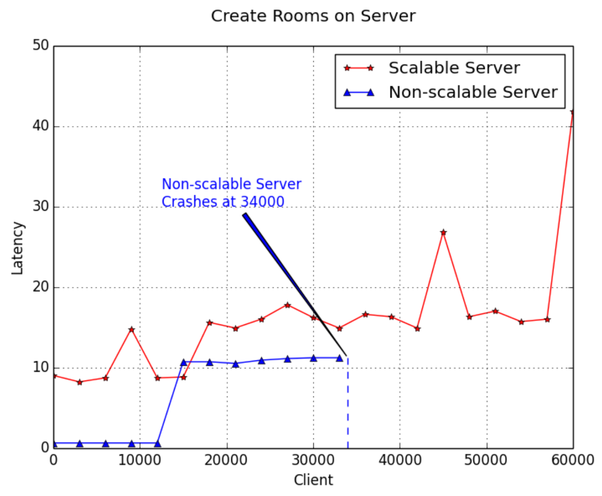

Fig. 2. Graph showing the average room creation latency against the number of new rooms created

	Stage	Ping / 1 sec	Ping / 2 sec
Median Lat. / ms	Stage 1	0.1	0
	Stage 2	0.6	0.1
	Stage 3	12.8	0.4
	Stage 4	63.4	6.1
	Stage 5		19.2
	Stage 6		44.2
P95 Lat. / ms	Stage 1	0.3	0.1
	Stage 2	16.7	0.7
	Stage 3	104.2	21.7
	Stage 4	246.4	55.8
	Stage 5		85.5
	Stage 6		177.8
P99 Lat. / ms	Stage 1	0.6	0.3
	Stage 2	31.4	3.7
	Stage 3	170.5	35.4
	Stage 4	337.8	123.9
	Stage 5		226.3
	Stage 6		

TABLE I
Table showing results from Artillery testing the WebSocket connection latency under different load

verifying the good concurrency performance of our backend system.

V. Further Work and Conclusion

In this report we have presented vChamber, a scalable cloud application that provides real-time synchronised video playback. Empirical results show that our back end server implementation has descent performance, and the whole system achieves some of our scalability goals, despite sacrificing latency. In this project we have experimented on designing and building a scalable low-latency cloud application with Kubernetes. The project itself might be further extended with more features, and some design choices of the system could be taken for future reference.

References

[1] “Syncplay.” [Online]. Available: https://syncplay.pl/
[2] Justineo, “Justineo/coplay,” Jul 2017. [Online]. Available: https://github.com/Justineo/coplay

[3] M. Kim and B. Noble, “Mobile network estimation,” in *Proceedings of the 7th annual international conference on Mobile computing and networking*. ACM, 2001, pp. 298–309.

[4] S. Potts, “A simple, customizable html5 video, audio, youtube and vimeo player.” [Online]. Available: http://plyr.io/

[5] “Prepare your systems for high load.” [Online]. Available: http://artillery.io/