Real-time Video Server with WebRTC

Mingzhou Yang

Master of Science

Department of Computer Science

McGill University

Montreal,Quebec

2015-11-23

A thesis submitted to McGill University in partial fulfillment
of the requirements of the degree of Master of Science

DEDICATION

This document is dedicated to the graduate students of the McGill University.

ACKNOWLEDGEMENTS

There are number of people without whom this thesis might not have been written. First of all, I would like to express my sincere gratitude to my supervisor Prof. Muthucumaru Maheswaran for the continuous guidance of my research and the writing of this thesis. Then I would like to thank my parents for educating and supporting me throughout my life. My friend Ahmed Youssef also helped me a lot to improve the grammar of this thesis.

ABSTRACT

Traditional Peer-to-Peer (P2P) video streaming services require particular software or browser plug-ins to manage peers and distribute content. However, with the emergence of Web Real-Time Communication (WebRTC), it becomes possible to accomplish browser-to-browser data exchange without any intermediate servers. In this work, we design and implement a real-time P2P video server based on WebRTC. Our server owns both the real-time feature of client-server transmission and the scalability feature of P2P protocols. We apply a simple algorithm to distribute content to peers and manage all the peers. Fault-tolerance mechanisms are also employed to improve the server's stability. We conducted several experiments on various features of both the server and the clients. Our work also demonstrates current limitations of our system and discusses features that will be added in the future.

ABRÈGÈ

Les services traditionnels de vidéo en streaming pair à pair (P2P) nécessitent un logiciel particulier ou des plug-ins de navigateur pour gérer les pairs et distribuer du contenu. Cependant, avec l'émergence de la communication web en temps réel (WebRTC), il devient possible d'accomplir des échanges de données de navigateur à navigateur sans serveur intermdédiaire. Dans ce travail , nous concevons et réalisons en temps réel un serveur vidéo P2P basésur le WebRTC. Notre serveur possède à la fois les caracteristiques en temps réel de la transmission client-serveur et la fonctionnalité de l'évolutivité des protocoles P2P. Nous appliquons un algorithme simple pour distribuer du contenu aux pairs et gérer tous les pairs. Les mécanismes de tolérance de panne sont également utilisés pour améliorer la stabilité du serveur. Nous avons mené plusieurs expériences sur diverses caractéristiques à la fois sur les serveur et sur les clients. Notre travail démontre dégalement les limites actuelles de notre système et discute les caractéristiques qui seront ajoutés dans l'avenir.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
Introduction

With consumers' increasing passion about watching live videos online instead of TV, video streaming service providers like Netflix, Youtube and Youku occupy an increasing percentage of Internet traffic. According to a report by Sandvine [1], Netflix and YouTube accounted for nearly half of the downstream traffic during peak hours in the second half of 2014. It is also predicted that 75 percent of all channels will be born on the Internet and 90 percent of Web traffic will be video in the next decade [2]. Considering the enormous bandwidth needed for video service providers, it has become a big issue for them to offer a great viewing experience as well as reduce network expenses.

In order to improve consumers' experience while watching videos, Content Distribution Network (CDN) is a solution. With the help of CDN, video service providers can store content in multiple locations. Then, requests for content will be redirected to the nearest CDN node in order to minimize transmission time and achieve high performance.

In order to reduce network expenses, Peer-to-Peer (P2P) protocols are considered to be the most feasible solution. Although there exist some successful commercial sites using P2P protocols like PPTV, CoolStreaming, PPStream and SopCast, P2P protocols are not adopted by large video service providers like Netflix, Youtube

and Youku because P2P solutions always rely on either client end software or browser plug-ins to coordinate P2P data exchange.

Web Real-Time Communication standard (WebRTC), drafted by the World Wide Web Consortium (W3C) in 2011, is a feasible method to achieve lightweight P2P data exchange. WebRTC is an API definition that supports browser-to-browser applications for voice calling, video chat, and P2P file sharing without the need for installing either internal or external plug-ins. Even though WebRTC is still under development, it has already drawn the interest of developers. Compared to other WWW multimedia technologies like Adobe Flash and Microsoft Silverlight, WebRTC has the advantage of being open source and potentially compatible. So, it is likely that, in the near future, WebRTC will provide interoperable multimedia communications and enable an effective convergence among desktop WWW platforms and Smartphones [3].

WebRTC implements three APIs: [4]

- RTCPeerConnection API is the WebRTC component that handles stable and efficient communication of streaming data between peers.
- MediaStream API represents synchronized streams of media (audio and video).
- RTCDataChannel API supports real-time communication for other types of data.

Here is a common scenario for how users set up a WebRTC connection. Firstly, peers need to register in a WebRTC connection server and acquire their unique IDs. Then, they talk to another public server to exchange their IDs with the help of the server. After being aware of each others' IDs, one of the peers sends a request to

the WebRTC connection server mentioned before with the target ID and the connection server coordinates the connection establishment process between the users with the help of RTCPeerConnection API. After that, users can communicate with each other using MediaStream API or RTCDataChannel API without any intervention from servers. Therefore, two servers are needed here: one for peer registration and connection coordination, and one for peer ID exchange. In our work, these two servers are located in the same machine.

Our WebRTC-based P2P Real-time System (WPRS) is also part of the Reality over Web (RoW) project [5] at the McGill Advanced Research Lab. RoW is a novel concept, where a window on the web corresponds to a window onto a real space. Once the correspondence is established, users should be able to interact or manipulate the objects or people in the real space through the web window.[6] As a result, our server is delay-sensitive so that the video can respond to user's manipulation quickly. In this way, instead of applying existing P2P video distribution algorithms whose delay time is unacceptable, we design a simple but effective algorithm for a small and medium number of users.

WPRS can be divided into two parts: the server end and the client end. In the server end, we implement a main server that processes captured frames and manages peers with C++. Peers are divided into several groups and, in a common scenario, all the peers in the same group receive different portions of data from the server. When peers connect with the server, disconnect from the server or are reported as slow by its groupmates, the server takes measures to manage the group and keep other

peers unaffected. We also run a WebRTC connection server to coordinate connection establishment.

Our client end program is totally implemented in JavaScript. After the client receives the packet including IDs of its group members, it sets up connections with its group members with the help of the WebRTC connection server. Whenever this client receives its portion of data from the server, it broadcasts data to all its groupmates. At the same time, it also collects data from its groupmates and after a certain period of time, the frames will be displayed on the screen. When a client notices any of its group members are slow, it reports the main server automatically to ensure its performance.

1.1 Thesis Contribution

WPRS can be seen as a hybrid system with both the real-time feature of client-server transmission mode and the scalability feature of P2P protocols. Compared with traditional P2P protocols who have a playback delay of dozens of seconds [6] [7], WPRS has a significantly better performance. The frames displayed on our clients' screens are at most 4 seconds behind the live video.

On the other hand, our server can support more users when comparing with the traditional client-server transmission mode. Although our server is not fully scalable, it still greatly improves the server's capacity.

By applying WebRTC to implement peer-to-peer data exchange, no client end software or browser plug-ins need to be installed. The client end application is totally implemented by JavaScript and everything clients need to do is open a web page. It brings clients convenience and makes our system light weight.

Besides, we adopt an imageHash algorithm to decide the similarity of consecutive frames. If a frame is similar to its previous frame, then a signal is sent instead of the frame data. This mechanism minimizes redundant frame data sent and saves some bandwidth.

Finally, due to the highly dynamic nature of the Internet, we design a group management mechanism to deal with frequent group change such as peer join, peer leave and slow peers. This mechanism protects other clients in a changing group from being severely affected.

1.2 Thesis Outline

This thesis includes 5 chapters. For easy reference, the chapters that follow are organized as below:

- Chapter 2 provides the background and related works of video distribution protocols, including tree-based protocols, gossip-based protocols and pull-based protocols. Research about WebRTC standard is also included.
- Chapter 3 introduces the design of our server and clients in detail. It includes how our video server processes frames captured by the video camera before distributing them to peers. Also, it will be demonstrated how the server and clients behave when a peer tries to connect with the server, disconnect from the server or is reported as slow by other peers. The fault-tolerance mechanisms employed to improve clients' watching experience will also be introduced.
- Chapter 4 shows the experiments done about multiple features of both the server and clients. A comprehensive analysis about the results will follow.

- Chapter 5 gives a conclusion on our topic based on the results from our experiments. We will discuss the current limitations of our video server as well as the improvements needed to be done in the future.

CHAPTER 2
Related Works

In this chapter, we provide the related works about P2P video distribution algorithms and WebRTC.

2.1 IP Multicast

Before talking about P2P algorithms, we introduce IP Multicast first, which was initially considered as the solution for video distribution. IP Multicast aims to use network infrastructure (typically network switches and routers) to replicate packets during the transmission and forward them to corresponding ports. In this way, the server only needs to send each packet once even if there are a large number of receivers. Although IP Multicast sounds feasible and has been standardized in 1986 [8], its deployment still remains confined. The main reason is that current IP Multicast model lacks technical supports for authorization, security, address allocation and network management [9]. The lack of commercial motivation for Internet service providers (ISPs) to offer infrastructural support is also an important reason.

2.2 Tree-based Protocol

As a result, researchers turned to application-layer solutions for video distribution. Chu et al. [10] proposed a scheme called End System Multicast. According to their work, end systems can self-organize into an overlay structure using a fully distributed protocol. Via the overlay structure, end systems participate in the multicast group communication and play the same role as routers in IP Multicast. In this way,

these multicast related features such as group membership, multicast routing and packet duplication can be implemented at end systems through unicast IP services.

Their work has proven it possible to construct an overlay network on top of a dynamic, unpredictable and heterogeneous Internet environment without relying on a native multicast medium. For small and medium-sized multicast groups, it is feasible to use this end system overlay approach to efficiently support all multicast related functionality. Although the overlay approach to multicast cannot avoid some delay in end system communication, the performance penalties are acceptable in the case of small and medium sized groups.

After the work by Chu et al. [10], more works have succeeded in building such tree-based overlay networks to spread packets from the source to all destinations. Biersack et al [11] have shown that the tree-based architecture can reach the expected performance in certain circumstances. The main focus is how to construct and maintain an efficient distribution tree among the overlay nodes.

- CoopNet [12] uses a central server to collect the information from all the nodes and manage the tree structure. As a result, a strong and reliable server is needed to carry the load.
- In contrast, NICE [13] and ZIGZAG [14] achieve logarithmic tree height based upon a hierarchical clustering and do not require any underlying topology information.
- Narada [10] constructs trees in a two-step process: meshes are constructed among participating members first and then spanning trees of the mesh are

constructed using well-known routing algorithms. This mesh-based approach is motivated by the need to support multisource applications.

However, there still exists some vital drawbacks which prevent these protocols from being put into use [15].

- Leave or crash behavior of the upper-level nodes often causes buffer underflow in a large population of descendants and the tree has to be reconstructed. Due to the highly dynamic nature of the Internet, users may suffer from frequent transmission outages.
- As the upload bandwidth of nodes is much less than that of the server and also varies with each other, there exists an unfair distribution of the available bandwidth for each node.

2.3 Gossip-based Protocol

Gossip-based protocols were also a popular solution to multicast message dissemination in P2P systems, inspired by the form of gossip seen in social networks. In a gossip-based protocol, when a node receives a packet, it sends the packet to some randomly chosen nodes if it is the first time this node receives that packet. Otherwise, this packet is simply ignored. This behavior continues until all nodes have this packet.

Among all the works related to gossip-based protocols, Xu et al. [16] designed a system that achieves most of the designated features of a gossip-based protocol:

- Peers show different behavior with the server;
- Peers contribute differently depending on their own bandwidth;
- Streaming capacity of the system grows dynamically;

- Each streaming session may involve multiple supplying peers.

They also solved the problems about how to distribute the media to multiple supplying peers in the same streaming session and how to amplify the system's total streaming capacity efficiently. The solution to the first problem is a video streaming distribution algorithm called OTSp2p, which can minimize buffering delay in the consequent streaming session. This algorithm is executed by the requesting peer. It assigns each participating supplying peer the same proportion of media data segments as the proportion of this peer's out-bound bandwidth to the total out-bound bandwidth. After computing the media data assignment, it initiates the peer-to-peer streaming session by notifying all the supplying peers of the corresponding assignment.

The solution to the second problem is a distributed differentiated admission control protocol called DACp2p. It achieves efficient system capacity amplification by differentiating between requesting peers based on their upload bandwidth. Each supplying peer individually decides whether or not to support requesting peer in a probabilistic fashion. The higher the out-bound bandwidth a requesting peer can contribute, the greater the possibility that it will be admitted. At the same time, the peers with low out-bound bandwidth should not starve. This mechanism can encourage requesting peers to contribute their truly available out-bound bandwidth to the peer-to-peer streaming system and make use of the upload bandwidth of most peers efficiently.

In addition, [17], [18], [19], [20], [21] and [22] are all pioneering works related to gossip-based protocols. Compared to tree-based protocols, gossip-based protocols

are more fault-tolerant and decentralized. However, the random push behavior also results in redundancy, which can be a vital problem for high-bandwidth streaming applications. In addition, several other questions need to be addressed. [23].

Membership - Membership is a vital issue in P2P protocols. Before a peer can request media data from supplying peers, it needs to acquire its own specific membership information. This *who knows whom* relationship is probably the most important issue to consider while designing scalable implementations of gossip-based protocols. In the original protocol [24], it is assumed that every peer knows the existence of every other peer. However, this scheme is not practical when deployed in large systems because the storage required to store the membership information and the extra load on the network to maintain consistent views of the membership grow linearly with the size of the system. As a result, a decentralized protocol providing each peer only with a partial view of the system is needed to improve scalability. Such an algorithm must trade scalability against reliability: small views growing sublinearly with the system size obviously scale better, while large views reduce the probability that peers become isolated.

Network awareness - Network topology is also an important issue in gossip-based protocols. Without considering network topology, it is possible that a peer always request media data from remote supplying peers, which results in low transmission efficiency and significantly limits the applicability of these protocols. Most solutions rely on a hierarchical architecture to reflect the network topology. This mechanism then ensures that most data packets are forwarded to peers within the same branch of the hierarchy and the transmission efficiency can be improved to

a large extent. However, organizing peers into a hierarchy in a dynamic and fully distributed manner is not straightforward and is still an active area of research.

Buffer Management - Every peer has a buffer with limited size to store received messages. These messages are forwarded a limited number of times to some random selected peers. The problem is, it is possible that the rate of new information production in the system may overwhelm these peers. That is, the buffer capacity of peers may be insufficient to ensure that every message is buffered long enough so that it can be forwarded a sufficient number of times to achieve an acceptable reliability.

Message Filtering - So far, the gossip-based protocols aim to spread messages to every peer in the system. However, it becomes complicated when different groups of peers have different interests. It is ideal if a peer has a higher probability to receive a message it is interested in than a message it does not want. But the deployment of an adequate filtering mechanism is not trivial and at least two issues need to be addressed. The first one is that in a decentralized system, it is not trivial for a peer to know other peers' interest. The other problem is, even when peer A knows that peer B is not interested in a message, it is possible that peer B still have to receive that message if peer B is a critical node for peer A to reach other peers that are interested in the message. These issues can be seen as a consequence of the brittleness of membership information. If every peer knows all other peers, including their interests, it becomes much easier to use a global algorithm to route messages to interested peers only.

2.4 Push-Pull Protocol

Before introducing the push-pull protocol, I would like to explain a pull-based protocol called Coolstreaming [25], which is considered to be the world's first successful large-scale peer-to-peer live video streaming (P2PTV) system [26].

Coolstreaming is called a pull-based protocol since all the packets need to be fetched based on receiver's own initiative. Thus, it is a receiver-driven approach. Coolstreaming is an efficient, robust and resilient system which is easy to implement. The core behavior of Coolstreaming is that every node requests needed chunks from its neighbours who own the chunks and also supplies data to its neighbours. To achieve this, Coolstreaming system has three key features: membership management, buffer map presentation and scheduling.

Membership management - Each node has a unique ID and a list of the IDs of other active nodes. When a node joins, it firstly contacts the original supplier, which is a dedicated server in most cases. The original supplier randomly chooses a node as the deputy node and redirects the newcomer to the deputy node, who will provide a list of nodes as the newcomer's potential neighbours. After joining the system, every node generates a membership message to announce its existence periodically. These messages are flooded to other nodes using the Scalable Gossip Membership protocol [27]. Upon a node's departure or failure, its neighbours also flood a message to other nodes to inform this news.

Buffer map presentation - Before transmission, the video stream is divided into segments of identical size and a Buffer map is employed in every node to represent

segments' availability. Every node exchanges its Buffer map information with its neighbours so that it knows which segment to request and supply.

Scheduling - Given the Buffer map of a node and its neighbours, the node uses a scheduling algorithm to decide the currently needed segments. Coolstreaming uses a heuristic algorithm here. It calculates the availability of each segment in its neighbours and the segments with fewer potential suppliers are assigned higher priority. If multiple suppliers exist then the one with the highest bandwidth is selected. This algorithm can meet the two key constraints: heterogeneous bandwidth for nodes and playback deadline for segments.

Although Coolstreaming achieved good video playback quality, there were still two main drawbacks in the earlier system. The first one is that peers sometimes suffer from unacceptable initial start-up delay because of the random peer selection process and per block pulling behavior. The other drawback is a high failure rate in joining a program during flash crowd. Due to these two drawbacks, Coolstreaming failed to meet the commercial system requirement.

Later, Li et al. [26] improved the original pull-based protocol and implemented a hybrid pull and push mechanism. In the new Coolstreaming system, segments are divided into several groups. If a node pulls a segment from its neighbour, then the neighbour automatically pushes the rest of the segment group to the requesting node. In this way, pull overhead can be greatly reduced. Besides, multiple servers are also deployed to reduce initialization time to less than 5 seconds. A multiple sub-streams scheme is also implemented to enable multi-source and multi-path delivery

of the video stream, which significantly improve the video playback quality and the system's robustness.

As our work aims at low-latency synchronous real-time video distribution, pull-based protocols are not a good fit for us. Instead, we divide peers into groups and peers exchange data inside their group using a push-based protocol. Every peer automatically pushes data received from the server to all the other group members.

2.5 WebRTC

WebRTC was drafted by W3C in 2011 and currently still under development by W3C [28] and the IETF [29]. As a result, few works have been done on this topic and only Google Chrome, Mozilla Firefox and Opera have implemented WebRTC specifications.

Among the limited number of works about WebRTC, Eriksson et al. [30] analyzed the fields that WebRTC can be utilized. They pointed out that WebRTC can enrich communication within a community or a vertical application such as e-health, education or social network, and enables faster deployment of massmarket communication services.

Loreto et al. [31] explained WebRTC's architecture, principle and APIs in detail. They also indicated that WebRTC does not have a congestion control mechanism at this time because IETF is still working on it. It is ideal to use only one single congestion control instance for audio, video and data. At the same time, WebRTC should be able to prioritize part of the transfer. Besides, WebRTC also brings some security threats when direct browser-to-browser communication is allowed. We must consider communications security as we do with other network protocols (such as

SIP) that allow for direct P2P communication. It is also essential to create a process to let users verify each other before the connection is established.

Rhinow et al. [15] implemented a simplified pull-based protocol which shares many key features of Coolstreaming and GridMedia. Their experiments show that WebRTC has the features to implement P2P protocols with a large set of clients. Nevertheless, they also demonstrate that it is not practical to implement a complete Coolstreaming protocol using WebRTC, due to its current limitations. In addition to the issuses mentioned in [31] and [32], three more problems exist. First of all, there is currently an interoperability issue among browsers, which results in implementation difficulties. Second, the browsers' integration of WebRTC is still in beta status and there exists some bugs, which may terminate web applications unexpectedly. Moreover, the WebRTC API does not yet offer functions for connection management and establishment. As a result, a second communication channel is necessary to establish a connection. In our work, we use our main server to help establish WebRTC connection.

2.6 Summary of Existing Protocols

Table 2–1 gives a summary of the P2P protocols mentioned above. Their concept, drawbacks and representative protocols are recalled.

Protocol Category	Concept	Drawbacks	Representative Protocols
Tree-based Protocol	End systems self-organize into a tree structure and perform multicast	1. Leave or crash behavior of upper-level nodes may affect many descendants 2. There exists an unfair distribution of the available bandwidth for each node	CoopNet, NICE, ZIGZAG, Narada
Gossip-based Protocol	Nodes forward newly received packets to randomly chosen nodes	1. Data redundancy 2. Membership 3. Network awareness 4. Buffer management 5. Message filtering	ostream, P2CAST, PROMISE
Push-pull Protocol	Nodes request needed chunks from neighbors with the chunks and supply data to neighbors	Does not support real-time video transmission	Coolstreaming, GridMedia

Table 2–1: Summary of P2P Protocols

CHAPTER 3
Design

In this chapter, we introduce how the WebRTC-based P2P Real-time System(WPRS) is designed in detail. WPRS is a hybrid server providing both the real-time feature of client-server transmission mode and the scalability of P2P protocols. As soon as a frame is captured, the server cuts this frame to smaller pieces and distributes these pieces to each client immediately to ensure low latency. Then clients exchange data received from the server with their groupmates so that they each obtain a complete frame. This chapter is divided into five parts: video stream pretreatment, common transmission scenario, peer join, peer departure and fault tolerance.

3.1 Video Stream Pretreatment

In this section, we discuss how to process the live video before it can be distributed to the users. The process includes two stages: frame cut and image hash.

Traditional P2P video distribution algorithms cut the whole video into small blocks and distribute these blocks to peers. However, since WPRS aims at low latency, the server cannot wait for the collection of the complete video, split the video and, at last, send the blocks. Instead, WPRS collects the video frame by frame and distribute parts of each frame to peers.

After the camera captures each frame, the server continues to cut the frame to pieces of equal size. The number of the pieces equals the maximum number of

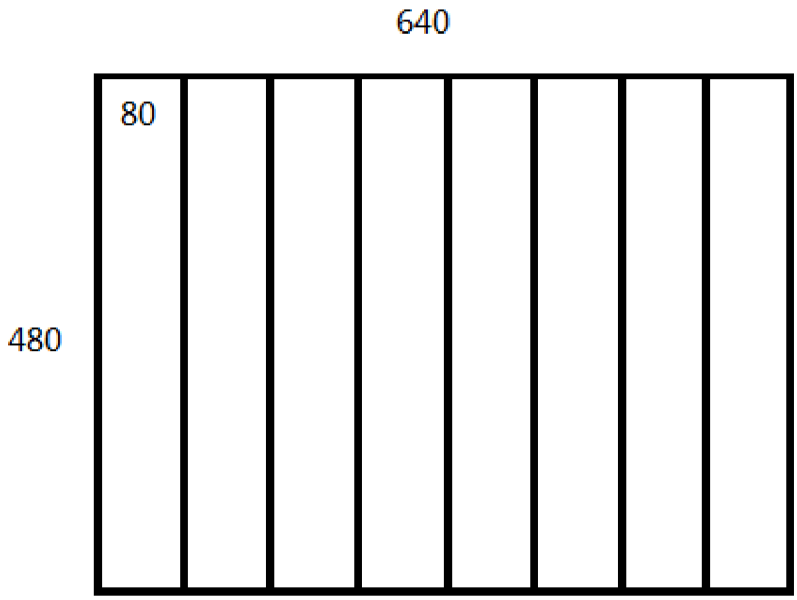

Figure 3–1: A frame with size 640 * 480 equally cut into 8 pieces.

peers in groups. For example, if we want to send a frame of size 640 * 480 to a full group with 8 peers, then the frame is cut into pieces of size 80 * 480, as shown in Figure 3–1.

We use the Open Source Computer Vision Library (OpenCV) to capture, store and cut frames. After a frame is captured, it is stored in a Mat object, which is a class with two parts: the matrix header containing its basic information and a pointer to the matrix containing pixel values. Then we can easily cut the frame into pieces using Mat's member function, given each piece's upper-left coordinate, height and width.

After that, an imageHash algorithm is applied to these pieces. This algorithm is adopted from an open source project called pHash [33]. The imageHash function

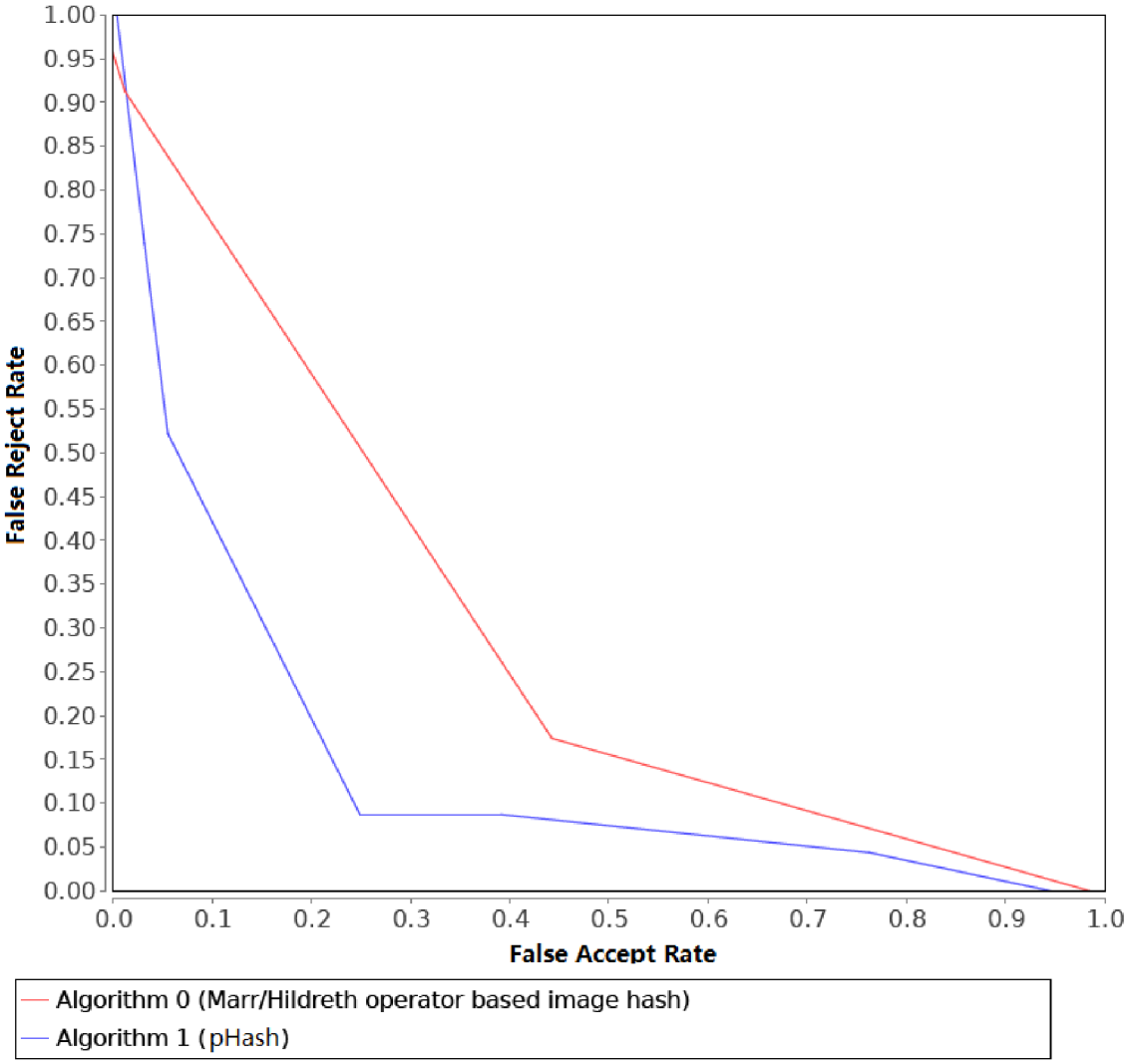

Figure 3–2: Performance comparison of pHash and another image hash function

takes an image as input and outputs a 64-bit integer. Unlike cryptographic hash functions which rely on the avalanche effect of small changes in input leading to drastic changes in the output, the imageHash function returns close hash values if the features are similar. Therefore, this algorithm can be used to test similarity of images.

When deciding whether two images are identical or not, we need a mechanism to compare their hash values. Let's denote their hash values by p_1 and p_2, then their discrepancy value is $|p_1/p_2 - 1|$. A discrepancy value close to 0 means that these two images are very similar. When the discrepancy value is less than a certain threshold, we can decide that these two images are identical.

For a given threshold, the performance of the imageHash function can be calculated based on the falsely classified images. Falsely classified images are either perceptually different images that are recognized as identical or perceptually identical objects which are recognized as distinct. These two kinds of misclassification can be represented by False Accept Rate (FAR) and False Reject Rate(FRR). For an ideal image hash function, both of its FAR and FRR should be close to 0. However, currently no algorithm can achieve it.

Zauners [34] used FAR and FRR as the benchmarks to measure the hash function's reliability. Figure 3–2 shows the performance comparison of pHash and another image hash function. The red line is another image hash function with worse performance and here we only focus on the blue line. We can see that the relationship of FAR and FRR value is like a reciprocal function. So it is reasonable to set the

threshold to a certain value that FAR and FRR values become equal. According to figure 3–2, the balance point is around 0.2 for FAR and FRR.

In our work, we firstly calculate the hash values of all the pieces of the frame. Then we compare these values with the values of the previous frame and check their similarity. If the difference between them exceeds the threshold, then this piece of frame needs to be transmitted to clients. Otherwise, the server only needs to send a DUPLICATE_PIECE signal, this piece's ID and the frame's sequence number to every client. This allows the clients to know that this piece did not change during the last frame and the piece from the previous frame is displayed on the screen again.

When a changed piece is recognized as unchanged, the video frame may be odd and the user's watching experience is influenced. On the other hand, if an unchanged piece is considered as changed, it only costs the server some extra bandwidth. Therefore, we prefer to pursue low FAR at the cost of a relatively high FRR and apply a conservative threshold. Depending on the experiments, we set the threshold to be 0.01 to ensure that the FAR is less than 5%.

The imageHash function's processing speed is closely related the size of the images. When processing original frames, the speed is so low that the server is overwhelmed. As a result, it is necessary for the server to resize the original frame to a smaller size before hashing. In this way, the time spent on imageHash becomes acceptable. This comes at a price of a reduction in accuracy. Besides, multiple threads are used here and each thread deals with one piece of the frame to accelerate imageHash.

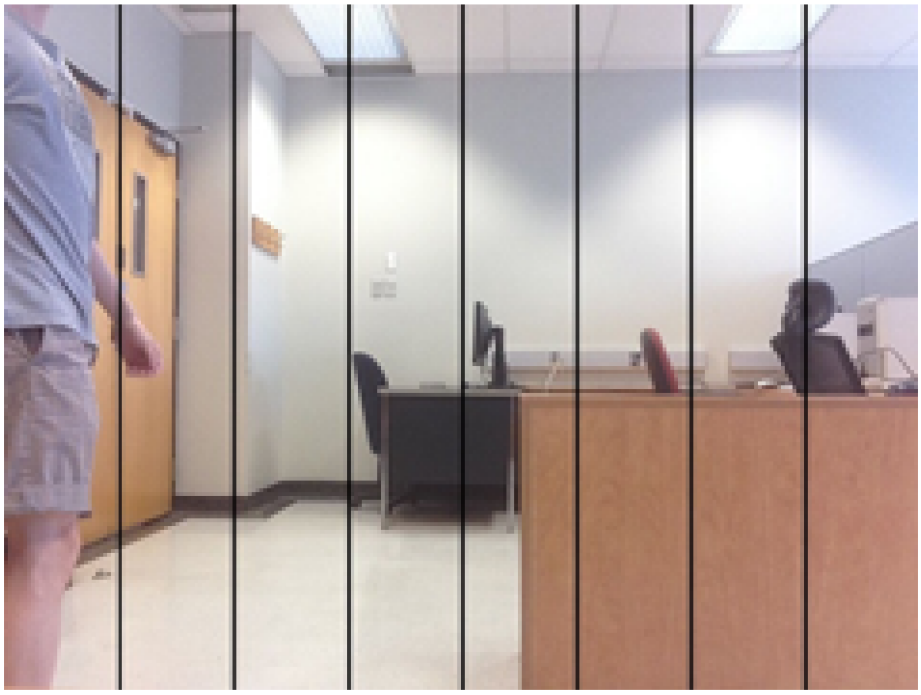

Figure 3–3: Pieces of Frame N

In our work, we choose frame pieces instead of frames as the smallest data unit. In this way, the imageHash function can be fully used to minimize redundant frame data sent. For example, if Figure 3–3 and Figure 3–4 are two consecutive frames, then only two leftmost pieces need to be sent to the clients. However if we consider every frame as the basic data unit and execute the imageHash function on these two complete frames, they will be judged as different frames. Then the whole frame in Figure 3–4 needs to be sent to the clients.

During the time when the imageHash function is being executed, there is also a thread which transforms the frame pieces to jpeg format and then stores them in a buffer. When imageHash is completed, the new hash values are stored in an array

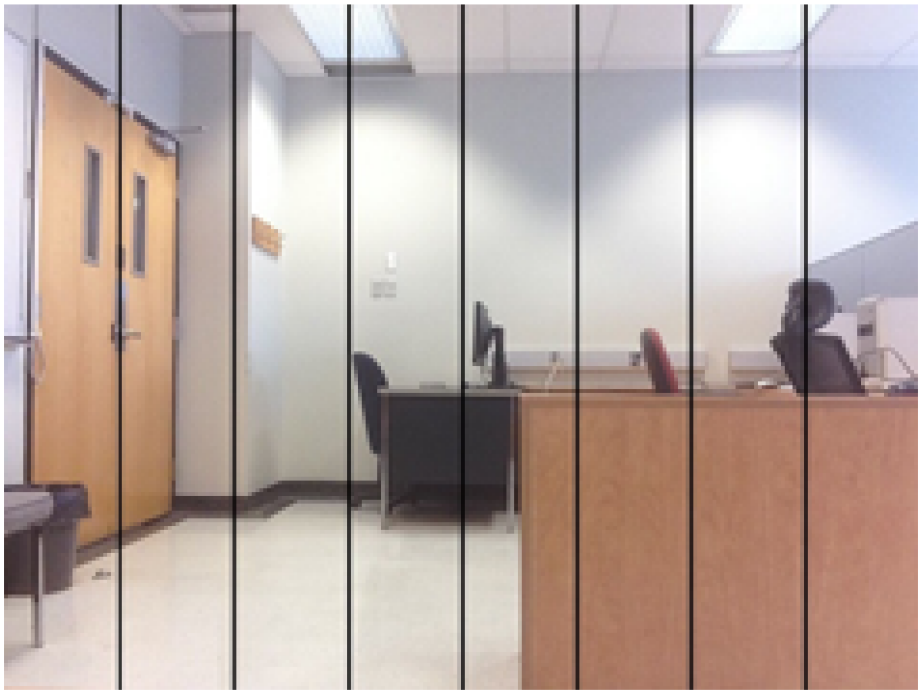

Figure 3–4: Pieces of Frame N+1

to replace the previous ones. Then, either frame pieces or the DUPLICATE_PIECE signal are sent to peers, based on the results of the imageHash computation.

3.2 Common transmission scenario

After the frame is preprocessed and stored, they are distributed to peers. Peers are divided into several groups with the same maximum group size. Every peer only communicate with its groupmates and the server, and does not know the existence of other groups of peers. Every peer receives at least one piece of each frame from the server. Recall that the number of the pieces in each frame equals the maximum number of peers in groups. So every peer in a full group receives exact one piece of the frame from the server. If a group is not full, then some peers may receive more than one piece of the frame. For example, if the frame is cut into eight pieces and

there exist six peers in this group, then two of the peers receive one more piece than other peers from the server. We later show how to decide the destination of each frame piece. It is guaranteed that there do not exist two peers in the same group receiving the same piece of the frame from the server. It is also ensured that every group receive all the pieces of each frame from the server, regardless of the group size.

When a client receives a piece of the frame from the server, it forwards its piece to all its groupmates using WebRTC. At the same time, each client has a listener waiting for packets from its groupmates. The received packets include not only frame pieces, but also this piece's frame sequence number and piece ID. Sequence number is the ID of the frames, which is sequential and increasing for all the frames. Piece ID is a non-negative integer less than maximum group size. For all the pieces of the same frame, their piece IDs are guaranteed to be unique.

Each client has a buffer called ReceiveBuffer to store frame pieces received from both the server and its groupmates. ReceiveBuffer behaves like a circular buffer of size is M*N, where M is the number of frames saved in the buffer and N is the number of pieces in each frame. Normally, N pieces stored in the same row belongs to the same frame. So each row in ReceiveBuffer can be displayed as a frame.

If all the frames are received in order, that is, all the pieces of $frame_i$ must be received before any piece of $frame_{i+1}$, then ReceiveBuffer can be seen as a real circular buffer. As a circular buffer, ReceiveBuffer has a write index indicating the row to store the next frame. Piece ID is used to determine the exact slot in this row to keep the piece. Assume the current write index is W and the received piece's

ID is P, this piece is stored in $ReceiveBuffer[W][P]$. There is also a read index indicating the row in which the frame to be displayed is stored. Whenever a frame is completely stored or displayed, the relevant index is increased by one. When either index reaches M, it is set to 0, which makes the buffer circular.

However, when considering the unpredictable and heterogeneous Internet environment, peers cannot expect to receive the frames in order. Thus, the sequence number received with the piece is needed to find the correct row to keep the piece. Assume that a piece's sequence number is S and piece ID is P, the piece is stored in $ReceiveBuffer[S \mod M][P]$. The module function used here makes the buffer circular.

Let's denote the largest sequence number received by S, then the read index of ReceiveBuffer is always set to $(S+2) \mod M$. It does not mean that the frame displayed is ahead of the latest frame received. Instead, it means that the latest frame is (M-2) frames ahead of the displayed frame. Let's denote the video's frame rate by F, then the approximate delay between the video captured by the server and the video displayed on clients' screen is (M-2)/F seconds, regardless of the transmission delay. In other words, after the client receive the first piece of the frame, it has (M-2)/F seconds to wait for other pieces before the frame is displayed. The reason why we do not set the read index to $(S+1) \mod M$ is that the frame stored in row $(S+1) \mod M$ is reserved for fault tolerance, which will be demonstrated later in detail.

We need to decide the value of M carefully because a large M means a long delay and more storage space. On the other hand, if M is not large enough, it is

more likely that clients fail to collect all the pieces before a frame is displayed. Based on the experiments, we set M to 50 because a larger value of M cannot bring smaller packet loss rate. At the same time, the length of the delay, which is about 4 seconds, is acceptable.

Figure 3–5 shows an example of ReceiveBuffer. If the received packet includes a piece whose frame sequence number is 703 and piece ID is 6, then it is stored in receiveBuffer[3][6]. The current frame displayed on client's screen is receiveBuffer[5] with frame sequence number 655, which was received several seconds ago.

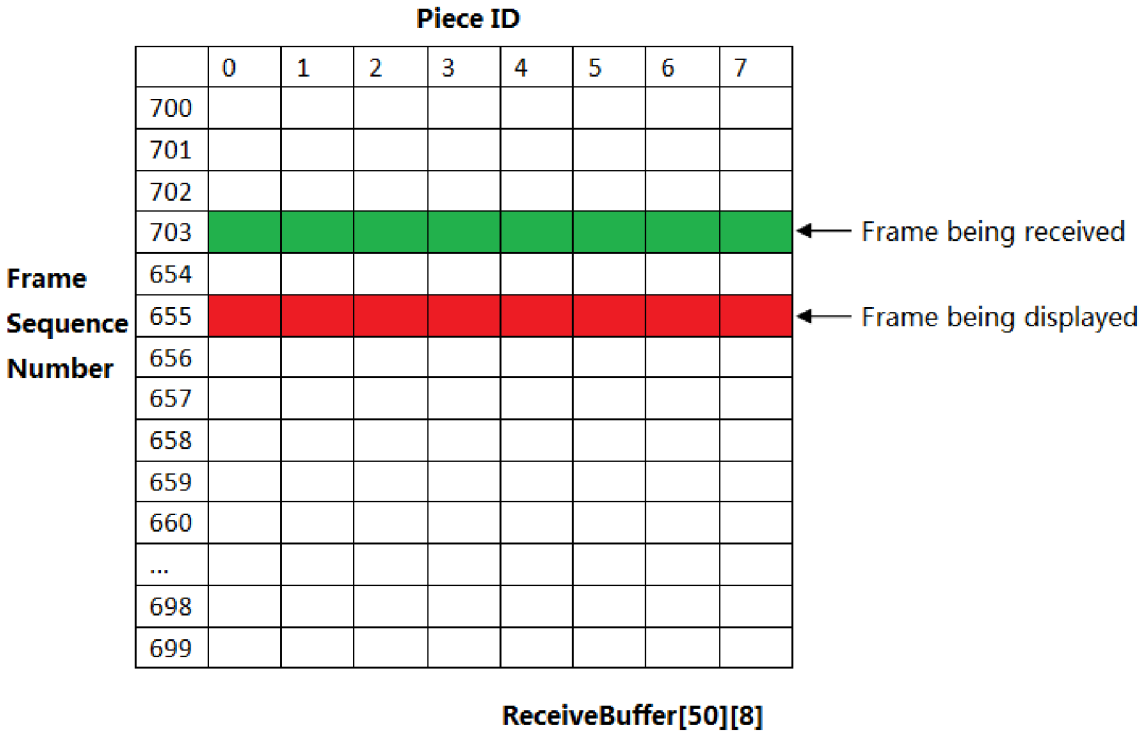

Figure 3–5: When frame 703 is being received, frame 655 is being displayed

If DUPLICATE_PIECE signal is received instead of frame pieces, we set $receiveBuffer[S \mod 50][P] = receiveBuffer[(S-1) \mod 50][P]$, where S and P are the piece's sequence number and piece ID. In this way, this piece from the last frame is displayed on screen.

When one piece of frame is replaced by a new piece, its memory needs to be deallocated to avoid a memory leak. However, as mentioned above, when DUPLICATE_PIECE signal is received, the URL of the previous piece is copied to the current slot. So it is possible that one piece is displayed in several frames. Deallocating a piece which is needed by other frames can result to a display failure. Therefore, before one piece is deallocated, we need to check whether this URL exists anywhere else in ReceiveBuffer. Only when no frame needs this piece can it be deallocated.

Figure 3–6 and Figure 3–7 show the transmission process of the server and clients in common transmission scenario.

3.3 Peer Join

Before introducing the behaviour of clients and the server when a new peer joins, we first describe the server's main data structures.

The server stores clients' information in a structure called Peer. Figure 3–8 shows all the attributes of Peer. Among them, **id** is the unique identifier that clients use to set up WebRTC connections with each other. According to the **mode** attribute, the server sends different packets to clients to inform group changes. There are four modes: NORMAL, GROUP_JOIN, GROUP_DELETE and RESPONSIBILITY_MAP. Besides, **peerToConnect** queue keeps IDs of the peer's new groupmates

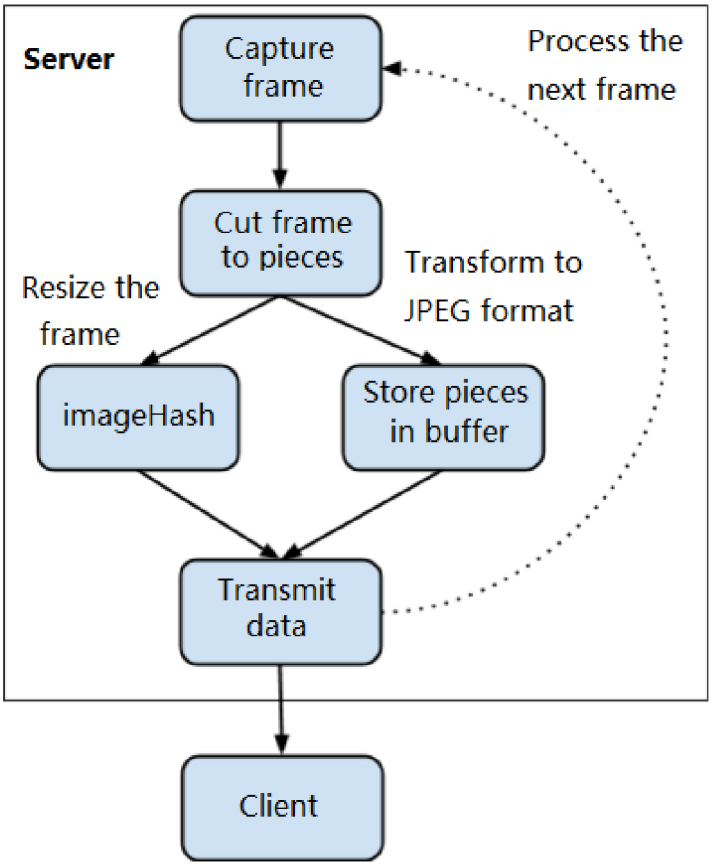

Figure 3–6: The server's process to send a frame

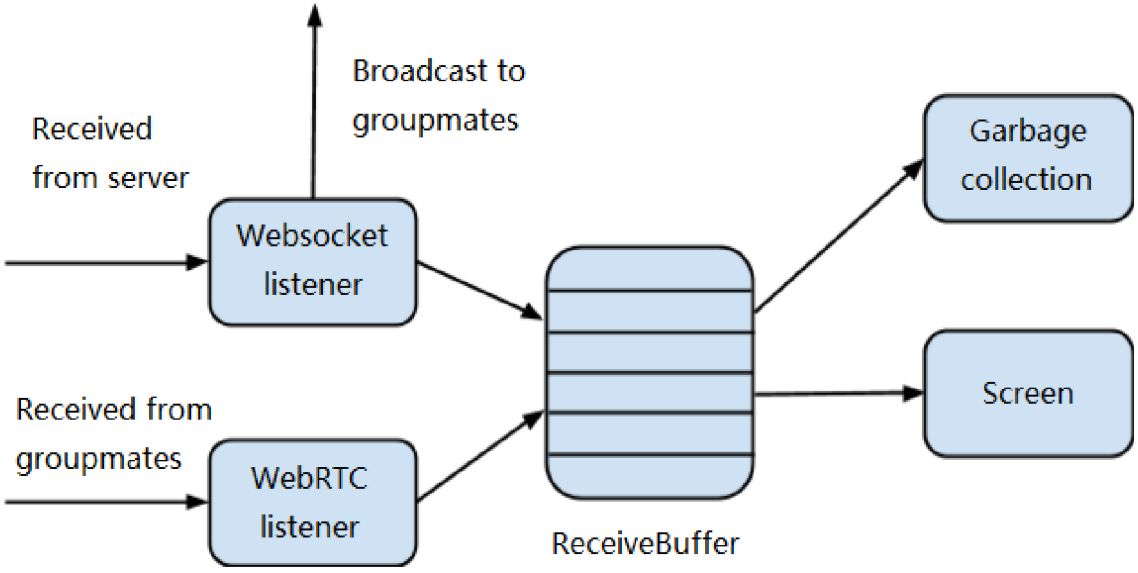

Figure 3–7: The client's process to exchange pieces with groupmates and display frames

and **pieceID** vector stores IDs of the pieces received from the server. These attributes and the server's various behaviour based upon these modes will be discussed later in this chapter.

```
struct Peer{
    string id;                  //Peer ID
    int groupNo;                //Group number of this peer
    int mode;                   //Behaviour mode of this peer
    queue<int> peerToConnect;   //Record peers to connect with
    vector<int> pieceID;        //Record pieces to receive
};
```

Figure 3–8: Attributes of Peer Structure

The server manages clients by a two-dimensional vector called GroupTable. Each row of GroupTable stands for a group. Groups need not be full. Figure 3–9 shows the structure of GroupTable with 4 groups. The integers in the table are the clients' identifiers.

Group 1	22	25	39	43	27			
Group 2	41	23	34	28				
Group 3	26	30	29	33	40	35	32	21
Group 4	44	24	38	31	35	36		

Figure 3–9: GroupTable with 4 groups

When a user enters the web page, the client end program starts, which is implemented in JavaScript. The client end program registers itself in the WebRTC connection server and is given a unique peer ID. Then the client end program tries to set up a WebSocket connection with the server through a hard coded IP address and port number. WebSocket is a protocol providing full-duplex communication

channels over a single TCP connection, which was standardized by the IETF as RFC 6455 in 2011 [35]. All communication between this client and the server will be done through the Websocket channel. After the connection is established, the client sends its peer ID to the server in order to join a peer group.

Algorithm 1 manageGroup

```
1:  procedure MANAGEGROUP(New_Peer)
2:      N ← −1
3:      for each group_i ∈ GroupTable do
4:          if group_i.size! = MAX_GROUP_SIZE then
5:              N ← i
6:          end if
7:      end for
8:
9:      if N == −1 then
10:         Create a new group G
11:         G.add(New_Peer)
12:     else
13:         for each peer_i ∈ GroupTable_N do
14:             peer_i.mode ← RESPONSIBILITY_MAP
15:             New_Peer.PeerToConnect.add(peer_i)
16:         end for
17:         New_Peer ← GROUP_JOIN
18:         GroupTable_N.add(New_Peer)
19:         REDISTRIBUTE(New_Peer, GroupTable_N)
20:     end if
21: end procedure
22:
```

As soon as the server receives the request, it calls the manageGroup function to put the new peer into a proper group. If no group exists or all the groups are full, then a new group is created and the new peer becomes the only member of the new group. Otherwise, the new peer is added to that group and its mode is

Algorithm 2 Redistribute_Join

```
1: procedure REDISTRIBUTE_JOIN(New_Peer, Group)
2:     max_num ← 0
3:     max_index ← −1
4:     for each peer_i ∈ Group do
5:         if peer_i.pieceNum > max_num then
6:             max_num = peer_i.pieceNum
7:             max_index = i
8:         end if
9:     end for
10:    for half of piece_i ∈ peer_max_index do
11:        peer_max_index.delete(piece_i)
12:        New_Peer.add(piece_i)
13:    end for
14: end procedure
```

set to GROUP_JOIN. The new peer will be aware of all its groupmates' peer IDs so that it can later set up WebRTC connections with them. All the other peers in the group are changed to RESPONSIBILITY_MAP mode. Algorithm 1 explains the manageGroup function in detail.

The next step is to redistribute the frame pieces so that the new peer can receive pieces from the server. All the peers in this group are checked and the one who receives the most pieces from the server is selected to hand out half of its pieces to the new peer. Algorithm 2 show more details about the redistribute algorithm.

Before sending frames, the server needs to check the mode of each peer. If a peer's mode is RESPONSIBILITY_MAP, a responsibility map is sent to this peer to inform the ownership of frame pieces after group member changes. Peers store the responsibility map for slow peer detection, which will be introduced later.

Otherwise, if a peer's mode is GROUP_JOIN, the server sends not only the responsibility map but also its new groupmates' peer IDs to the peer. As mentioned above, only the newcome peer's mode is set to GROUP_JOIN. That is because according to WebRTC connection mechanism, only the peer who requests for the connection needs to know its groupmates' peer IDs. Peers waiting for the connection do not need to be aware of their groupmates until the connection is established.

As soon as the newcome client receives its groupmates' peer IDs, it sends a connection request to the WebRTC connection server with its groupmates' peer IDs. This server stores all the peers' registration information and coordinates connection between peers. After the connection is set up, the newcome client can receive data from both the server and its groupmates. After that, all these peers return to NORMAL mode.

Figure 3–10 shows the overall process of peer join behaviour.

3.4 Peer Departure

When a peer leaves, the server can detect the Websocket connection is disconnected. In this case the manageGroup function is called to delete the departed peer from its group. This process is similar to peer join process. Firstly, all its groupmates are set to GROUP_DELETE mode. The peer with the least pieces in the group receive all the pieces from the departed peer. After that, the peer is removed from GroupTable. If the group is empty, then the whole group is deleted.

Afterward, the mergeGroup function is called to check if there exist two groups that can be merged together. The mergeGroup function iterates all the peer groups and finds the group with the least group members. If the sum of the size of the found

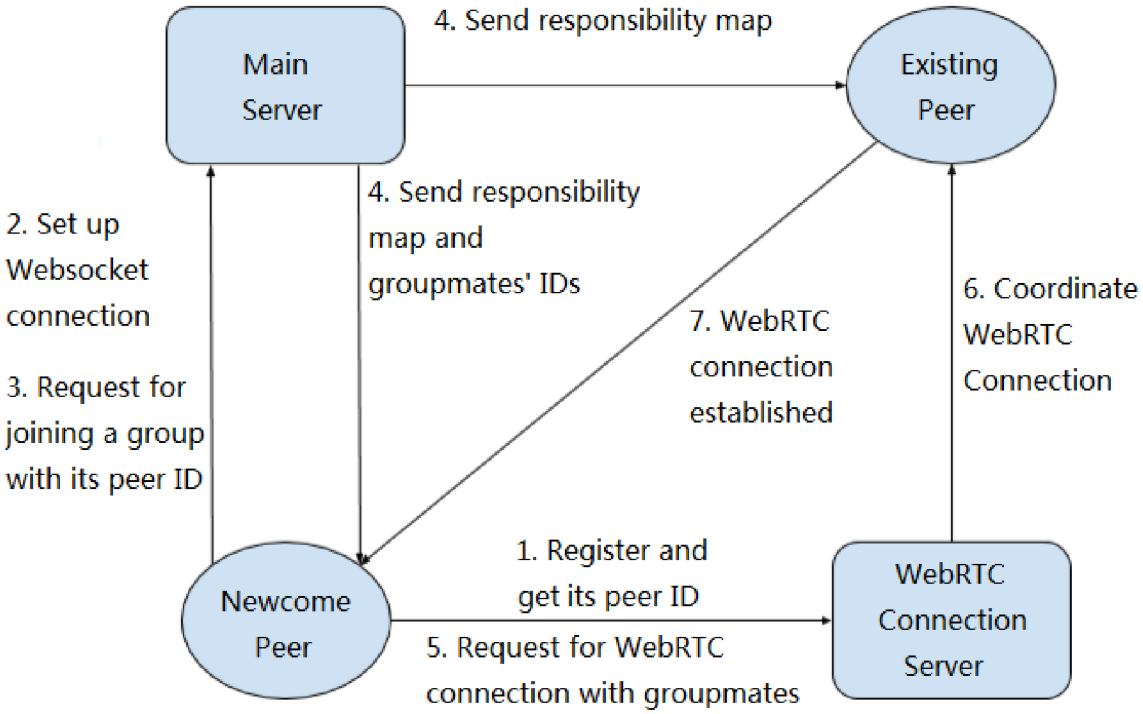

Figure 3–10: Flow Chart of Peer Join Behaviour

Algorithm 3 mergeGroup

```
1: procedure MERGEGROUP(Group)
2:     min_size ← MAX_GROUP_SIZE
3:     N ← −1
4:     for each group_i ∈ GroupTable do
5:         if group_i.size < min_size then
6:             min_size ← group_i.size
7:             N = i
8:         end if
9:     end for
10:    if Group.size + min_size <= MAX_GROUP_SIZE then
11:        for each peer_i ∈ Group do
12:            for each peer_j ∈ group_N do
13:                peer_i.peerToConnect.add(peer_j)
14:                peer_i.mode ← GROUP_JOIN
15:            end for
16:        end for
17:        for each peer_j ∈ group_N do
18:            peer_j.mode ← RESPONSIBILITY_MAP
19:            group_N.delete(peer_j)
20:            Group.add(peer_j)
21:        end for
22:        GroupTable.delete(group_N)
23:        REDISTRIBUTE(Group)
24:    end if
25: end procedure
```

Algorithm 4 Redistribute_Merge

```
1: procedure REDISTRIBUTE_MERGE(Group)
2:     Piece_Per_Peer ← MAX_GROUP_SIZE/Group.size
3:     Extra ← MAX_GROUP_SIZE mod Group.size
4:     id ← 0
5:     for each peer_i ∈ Group do
6:         peer_i.pieceID.clear()
7:         if i < Extra then
8:             for j ← 0, Piece_Per_Peer + 1 do
9:                 peer_i.pieceID.add(piece_i d)
10:                id++
11:            end for
12:        else
13:            for j ← 0, Piece_Per_Peer do
14:                peer_i.pieceID.add(piece_i d)
15:                id++
16:            end for
17:        end if
18:    end for
19: end procedure
```

group and the group with the departed peer does not exceed the maximum group size, then these two groups are merged together.

Let's call these two groups being merged group A and B. To merge them, all the peers in group A are set to mode GROUP_JOIN and all the peers in group B are pushed to their peerToConnect queues. That is, every peer in group A requests for a WebRTC connection with every peer in group B. As all the peers inside group A and B have already set up a connection with each other, they do not need to connect again. Then all the peers in group B are set to mode RESPONSIBILITY_MAP. After that, all the peers in group B are moved to group A in GroupTable and the empty group B is deleted. The detailed algorithm to merge two groups is explained by Algorithm 3.

The next step is to redistribute the pieces. Since the group has multiple new members, the redistribution mechanism applied here is different from the one used when only one peer joins. All the pieces need to be totally redistributed. If the maximum group size is S and the new group has P peers, then ($S \mod P$) peers are assigned ($S/P + 1$) pieces and rest peers receive (S/P) pieces, as shown in Algorithm 4.

Before sending frames, these peers in mode GROUP_DELETE or RESPONSIBILITY_MAP receive the responsibility map from the server so that their local map can be updated. Peers in mode GROUP_JOIN also receive its new groupmates' peer IDs. Then all these peers return NORMAL mode again.

The goal of merging groups is to decrease the number of groups while affecting as few peers as possible. As mentioned, every group needs to receive a complete frame. So the network traffic is roughly $number\_of\_group * frame\_rate * average\_frame\_size$. Hence, merging groups can directly decrease the server's network traffic.

3.5 Fault Tolerance

Due to the highly dynamic nature of the Internet, it is common that some of the clients may be in a poor network condition and fail to provide data to their groupmates on time. Furthermore, as our system is delay-sensitive, we employ unreliable data channels for inter-group communication, which can result in packet loss. In order to increase the server's stability and the client's performance, several measures are taken.

3.5.1 Display Previous Piece

The first thing that needs to be done is that the client end program should be able to detect unreceived packets. Here, a buffer called SequenceNum is used, whose size is exactly the same as ReceiveBuffer. SequenceNum[i][j] stores the sequence number of the frame piece saved in ReceiveBuffer[i][j]. By iterating the SequenceNum buffer, we can easily distinguish updated pieces from outdated pieces by comparing their sequence numbers. For example as shown in Figure 3–11, before ReceiveBuffer[5] is displayed, all the sequence numbers in SequenceNum[5] are checked. Then we can find that the most updated pieces' sequence numbers are 655, while two outdated pieces' sequence number are 605.

The most straightforward way to deal with this condition is to display the previous piece. That is, ReceiveBuffer[4][3] is copied to ReceiveBuffer[5][3] and ReceiveBuffer[4][7] is copied to ReceiveBuffer[5][7]. If the pieces stored in ReceiveBuffer[4][3] and ReceiveBuffer[4][7] were updated and these two consecutive frames are similar, then the user may not notice packet loss.

However, this measure has several limitations. If the peers forwarding piece 3 and piece 7 always fail to send pieces on time, then the previous piece may also be outdated. In addition, if frame 655 changes greatly, the user can also find the whole frame to be odd with a piece from the last frame and the user's watching experience is affected to some extent. So displaying previous piece is a very basic and expedient measure to improve performance.

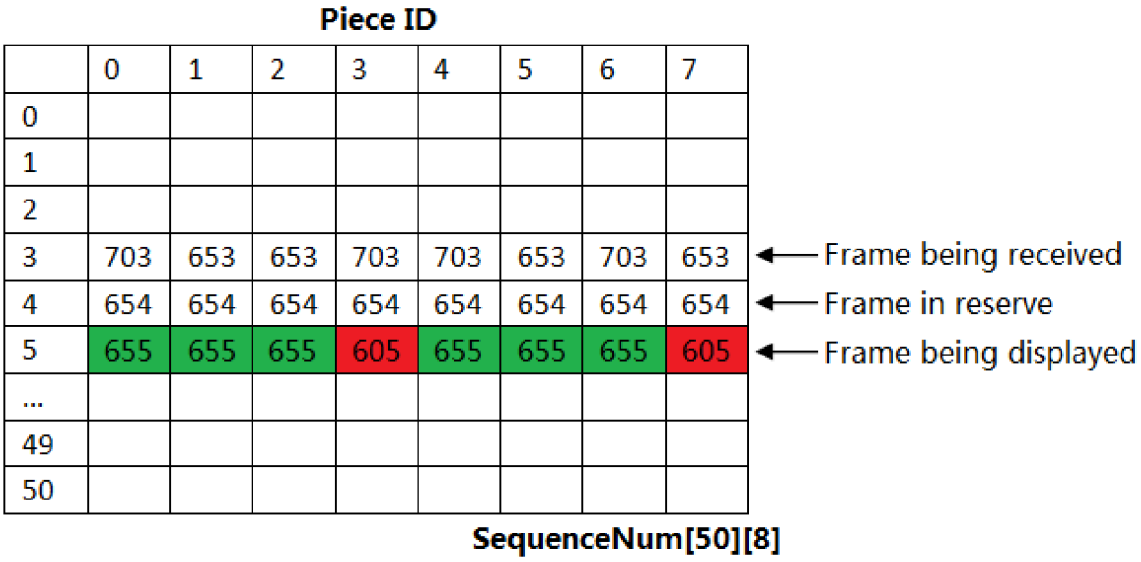

Figure 3–11: When frame 655 is being displayed, piece 3 and piece 7 are outdated

3.5.2 Round-Robin Sending

The second measure is taken by the server and can refine clients' experience even if one client is consistently slow. The server employs a round-robin algorithm to send the pieces. The peer who receives piece A will receive piece $((A + 1)$ mod $maximum\_group\_size)$ of the next frame. In this way, for each piece, the client who receives this piece keeps changing. Figure 3–12 and Figure 3–13 shows the piece distribution of two consecutive frames.

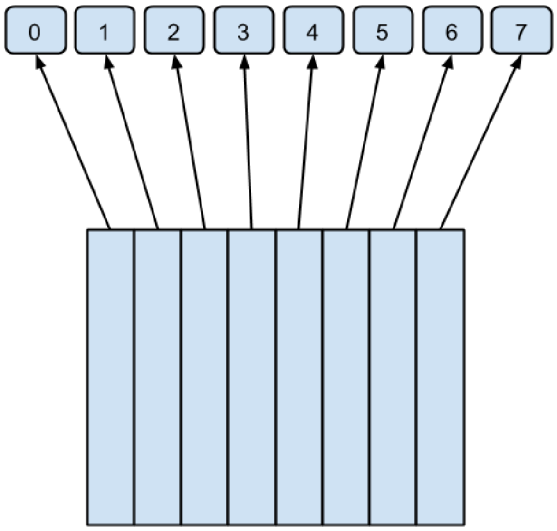

Figure 3–12: Piece Distribution of Frame N

By using this mechanism, even if one peer is consistently slow and fails to forward its pieces, the pieces influenced by it is always changing. At this time, if a piece is found to be outdated, then the previous piece, which is transmitted by another peer, can be displayed. If all the other peers in this group are in good condition, then the negative influence resulted from the slow peer can be minimized.

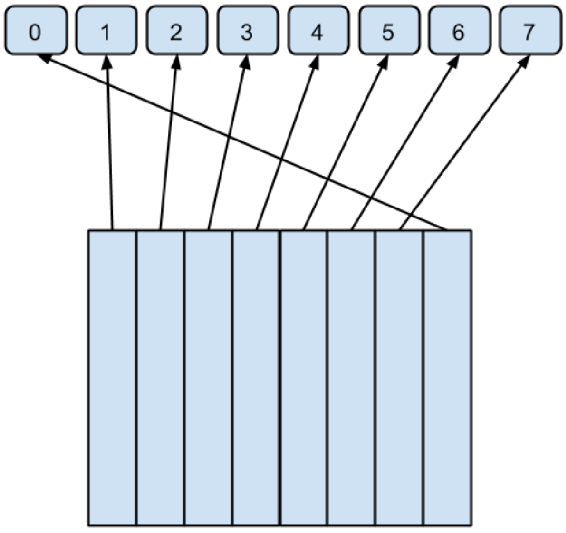

Figure 3–13: Piece Distribution of Frame N+1

However, round-robin sending mechanism is still only a palliative method to improve the performance. When there is a consistently slow peer, other clients in this group are still affected to a certain degree and may notice its existence if the frame changes greatly. Furthermore, if there are more than one slow peers in a group, the previous piece may also be received from a slow peer. Under these conditions, these two measures are hardly useful.

3.5.3 Slow Peer Handling

To completely solve the problem of slow peers, the server needs to remove these slow peers from normal groups to thoroughly eliminate their negative effect.

The first step is to identify these slow peers. Here we need the responsibility map received from the server. Responsibility map includes current frame's sequence number and the ownership of all pieces of this frame. When a piece is not received

on time, then we can find out the peer who is in charge of this missing piece by the responsibility map.

As mentioned above, the server sends pieces in a round-robin way, so some calculation is needed to find the responsible peer. Let's denote the number of pieces each frame includes by N. If the lost packet's sequence number is S1 and its piece ID is P1, while the sequence number of the responsibility map is S2 and the slow peer was responsible for piece P2 according to the responsibility map, then we can have this equation: $(P2 + S1 - S2) \mod N = P1$. This equation means that the peer who was in charge of piece P2 is now responsible for piece P1, after $(S1 - S2)$ frames are received. After calculation we can have $P2 = (P1 + S2 - S1) \mod N$. Finally, according to the responsibility map, the peer who was responsibility for piece P2 is the slow peer.

However, a peer should not be simply judged to be slow whenever it fails to transmit a packet on time, considering that packet loss is common when using unreliable data channels. Thus, each client end program maintains a buffer to keep the slow values of its groupmates, which is used to measure how slow these peers are. When a peer is responsible for a missing piece, its slow value is increased by 10. On the contrary, if a piece is successfully transmitted, its slow value is decreased by 1. In other words, a 10% packet loss rate is considered to be acceptable, as observed in experiments. As soon as a peer's slow value exceeds a certain threshold, this peer is reported to the server as a slow peer.

When the server receives a slow peer report, this peer is not considered to be slow arbitrarily. It is a possible situation that a peer with low download speed cannot

collect data from its groupmates on time and reports all its groupmates. To avoid mistaking normal peers for slow peers, only when half of the group members report a peer can this peer be judged as slow.

If a peer is determined as a slow peer, it is removed from its group immediately because it is not able to provide data to its groupmates on time. Indulging slow peers in the group can only ruin its groupmates' watching experience. However, removing a slow peer from its group does not mean that it is abandoned. It is put into a slow group instead. All the peers in the slow group can receive full frames from the server and do not need to provide data to any other peers. The transmission mode in the slow group is the simplest client-server mode.

However, since slow peers receive full frames from the server, supporting these slow clients may cost much more bandwidth than the normal clients. It is possible that the server spends too much resources on the slow peers, which is inadvisable. These normal clients should have higher priority than the slow peers. Thus, when the server do not have enough bandwidth, it sends frames with a lower resolution as well as lower frame rate to slow clients.

Considering the situation that a peer may be only temporarily slow, after a certain period of time a peer is put into the slow group, it will be given the second chance and moved to a normal group again. But if it is reported by its groupmates again, it will stay in the slow group forever.

CHAPTER 4
Evaluation

In this chapter, we conducted several experiments to evaluate our system's performance and capacity. The main yardstick we apply to measure the system's performance is the client's packet loss rate. High packet loss rate means the peer group or the server is in an unhealthy condition. In the following sections, we will demonstrate how factors like total peer number, maximum group size, imageHash function and group consistency impact clients' packet loss rate. It is also shown that how pretreatment time, group management time and transmission time change with increasing number of peers. Besides, the server's capacity of both normal peers and slow peers is measured as well.

4.1 Server Speed Test

Figure 4–1 shows how pretreatment overhead, group management overhead and transmission time vary when the total number of peers is increasing, with and without the imageHash function. Pretreatment overhead includes the time spent to capture frame, cut the frame to pieces and imageHash. Group management overhead is the time to add new peers to the groups, delete departed peers from the groups, merge groups and redistribute the pieces.

From the graph, we can see that when the imageHash function is implemented, much more pretreatment time is consumed, even though multiple threads are already applied and the frame is resized to accelerate the imageHash function. Due to the

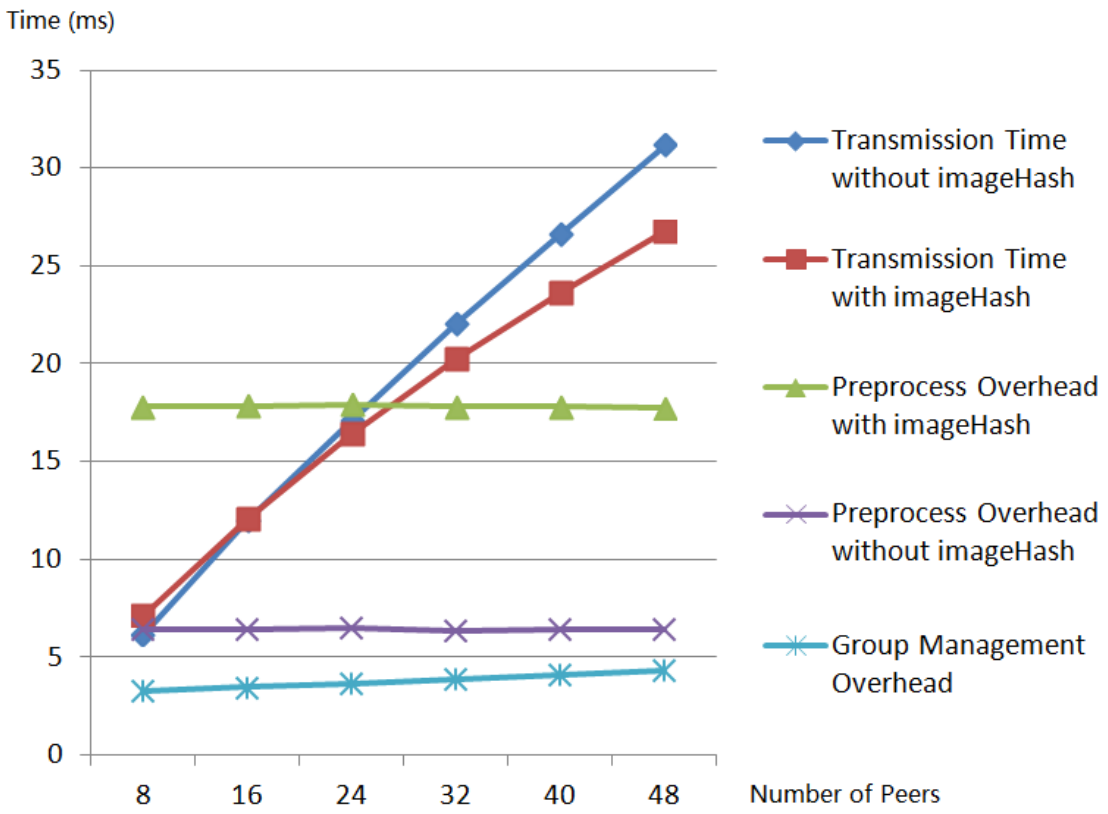

Figure 4–1: Pretreatment overhead, group management overhead and transmission time change with the number of peers, with/without imageHash

imageHash function, the pretreatment overhead also becomes slightly more floating. The pretreatment time barely change with peer quantity because the time needed to preprocess the frames only relate to the frames. With a growing size of groups, group management overhead slightly increases. There is only one curve of group management overhead because it is not affected by imageHash at all.

Transmission time always increases with the quantity of peers as the time needed to send each piece of frame remains while the number of pieces sent raises with the peer number. However, when the imageHash function is applied, the curve's slope is not so sharp as the one without the imageHash function because the imageHash function can prevent similar pieces from being sent again so that the total number of pieces sent is reduced to some extent. Thus, considering that imageHash itself is time consuming, only when the number of peers exceeds a certain value can we determine that the imageHash function is rewarding. To examine the effect of imageHash, clients' packet loss rate is a more straightforward measure, as shown in Figure 4–2.

4.2 imageHash Function Performance Test

Figure 4–2 demonstrates how packet loss rate changes with the number of peers, with and without imageHash function. It is mentioned in the last chapter that we use the responsibility map to detect the missing pieces of the frames. Hence, by counting the number of missing pieces and received pieces, we can calculate the packet loss rate.

It can be seen that both packet loss rate curves increase with peer quantity. The reason is that when there exist more peers, the server spends more time on transmission, as shown in Figure 4–1. However, since the server has to finish a

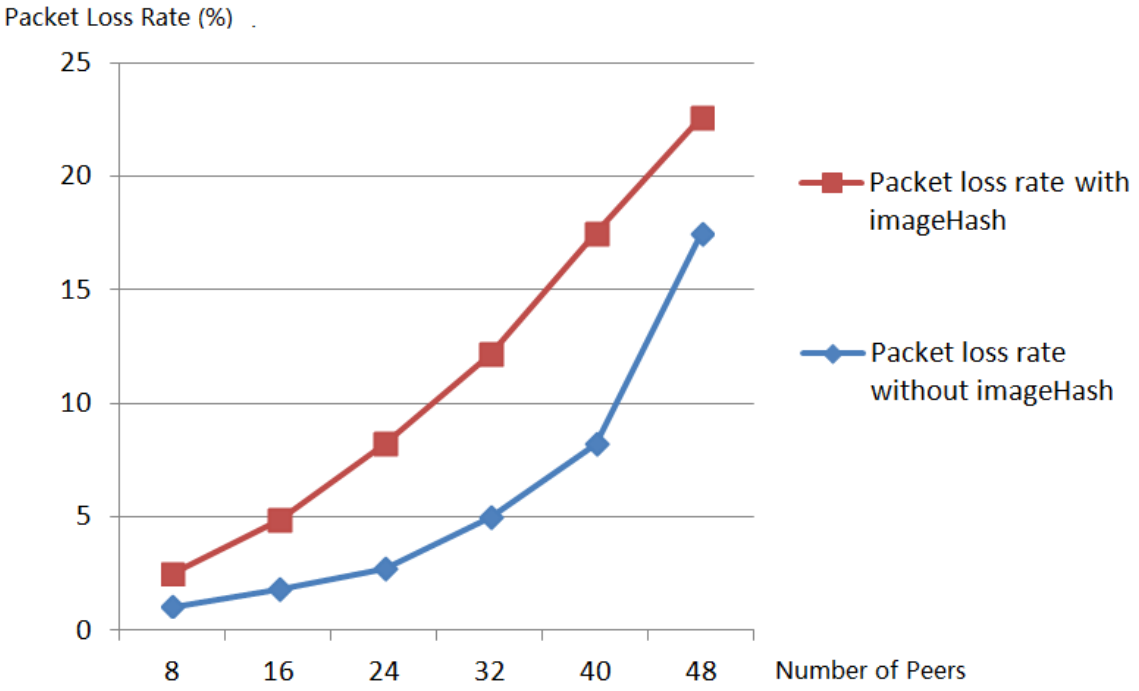

Figure 4–2: Packet loss rate changes with the number of peers, with/without imageHash

complete process of pretreatment, group management and transmission before the next frame is captured, the server processing is very time-sensitive. As a result, when the time spent on transmission approaches a certain threshold, the server is overwhelmed and can not respond to clients in time. That is why the curve without imageHash suddenly becomes sharp after 40 peers.

However, it is surprising that the curve with the imageHash function is above the one without imageHash. Since the imageHash function can reduce the number of pieces sent to some extent, the expected result is that the curve with the imageHash function may have a higher initial value but it can perform better when the quantity of peers grows. The most likely reason for the poor performance is that the resources needed for the imageHash function is way beyond expectation. Even if the imageHash function decreases the transmission time, it fades next to the massive time spent on the imageHash function itself. As a result, we can draw the conclusion that the bottleneck for the server at this moment is processing time instead of bandwidth. An efficient imageHash algorithm is needed to improve the server's performance and capacity.

Moreover, it should also be taken into consideration that even if the similar pieces are not sent, the server still needs to send DUPLICATE_PIECE signal to inform clients. In this way, despite bandwidth can be saved to some degree, the server still has to communicate with clients through the reliable data channel, which can cost some time. In addition, recall that in order to reduce the probability that a changed piece is mistakenly considered as unchanged, a conservative threshold is set. Thus, many unchanged pieces are regarded as changed and are still transmitted.

As mentioned in the last chapter, a 10% packet loss rate is considered to be acceptable. Thus, the server's capacity of normal peers is 40 if the imageHash function is not applied. When the imageHash function is deployed, no more than 24 normal clients can be supported by the server.

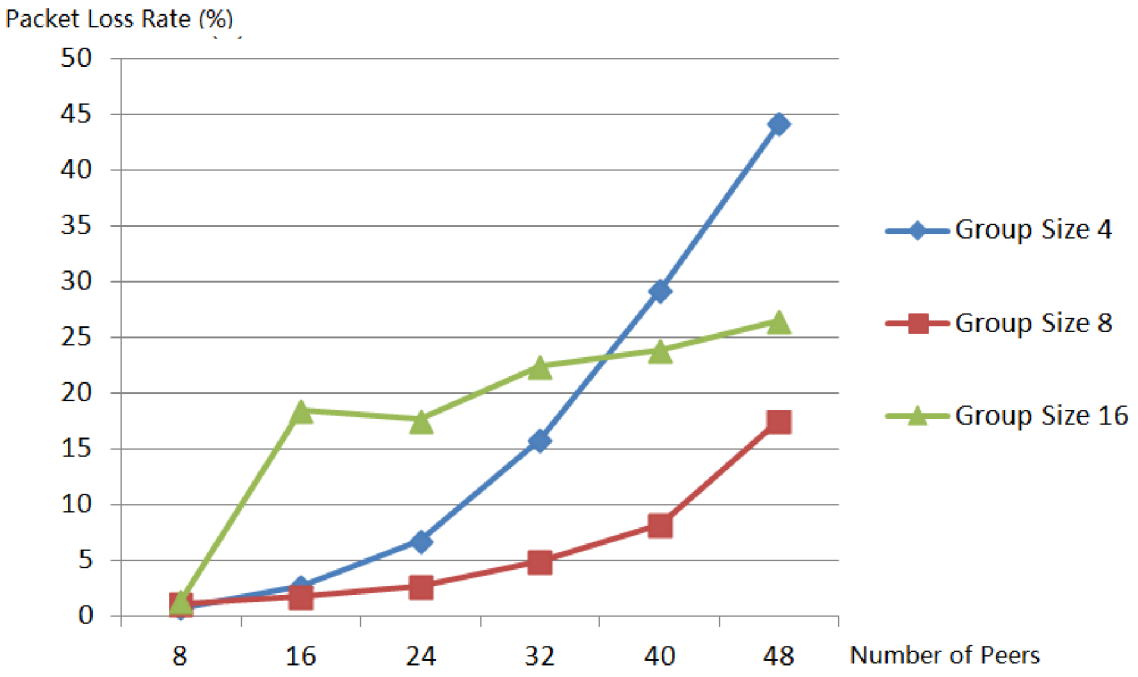

Figure 4–3: Packet loss rate changes with the number of peers, when maximum group size is 4, 8 or 16

4.3 Group Size Test

Figure 4–3 shows the difference among packet loss rate curves when maximum group size is 4, 8 or 16. As the imageHash function is resource-consuming, we want to avoid its influence in this experiment. Thus, the imageHash function is not applied here.

From Figure 4–3 we can see that the shape of the curve whose maximum group size is 4 is similar to the one of size 8. The blue curve has a steeper slope than the red one because, for the same number of peers, it needs to send twice as much data as the groups of size 8. As a result, it takes more time to transmit data and more bandwidth is occupied. Even if a smaller group can ease clients' burden of P2P data transmission, it can not counteract the negative effect due to the overwhelmed server.

However, the trend of the green curve seems intricate. It has a sharp rise at the beginning, drops a little bit and then ascend again slowly. The reason of the sudden rise lies in the fact that the group is too large. Every peer has to send its piece to 15 other peers and process pieces received from its groupmates in a short time. Therefore, the client end program is overwhelmed and is not able to transmit data to its groupmates in time. When there exist only 8 peers, they behave exactly the same as the red curve because the group is not full. However, when the peer number reaches 16 and a full group comes into being, the factors mentioned above increase packet loss rate by a big margin. Then 8 more peers connect with the server and constitute a new group. Since the new group is only half full, peers in this group can work properly and do not suffer from large packet loss rate. As a result, the average packet loss rate of all the peers is lowered to some extent. After that, with an increasing number of peers, the server is also gradually overwhelmed by transmission so that packet loss rate keeps rising.

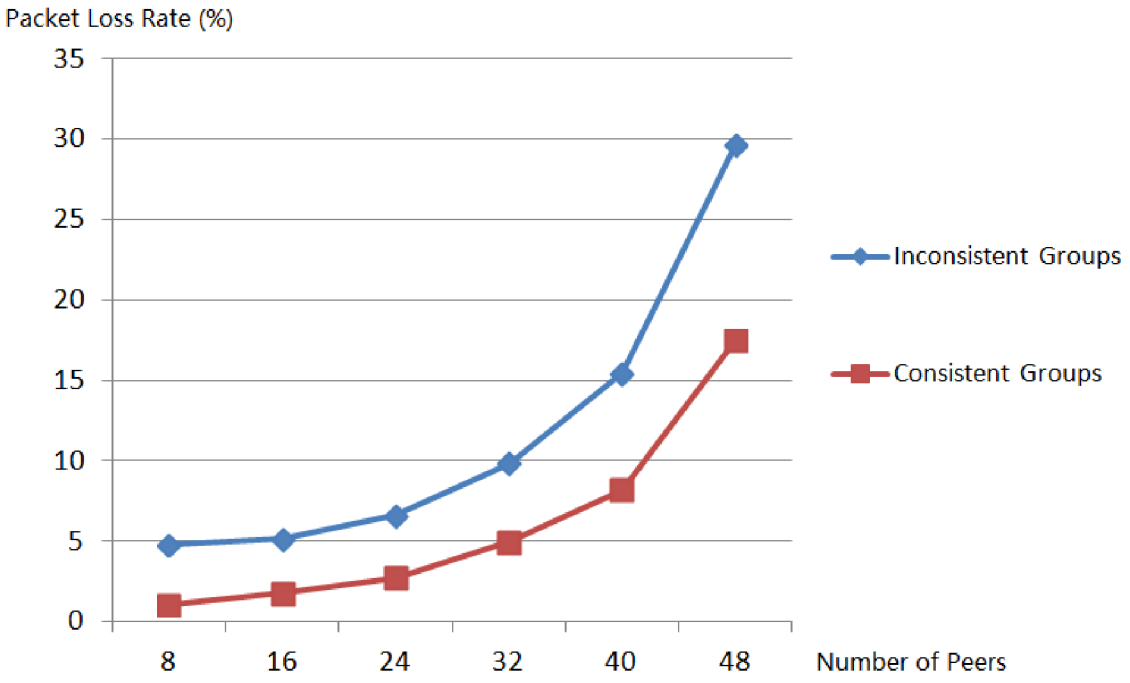

Figure 4–4: Frame rate changes with the number of peers, when groups are consistent/inconsistent

4.4 Group Consistency Test

Figure 4–4 compares the packet loss rate between consistent groups and inconsistent groups. For all the other experiments in this chapter, peers are assumed to keep staying in the group once the connection is established and never leave. As a result, those experiments were performed under ideal conditions. However, due to the highly dynamic nature of Internet, it is important to see how frequent peer join and leave can affect the system. In the experiment, every 10 seconds an old peer leaves the group and then a new peer joins the group.

According to Figure 4–4, we can see that peers in inconsistent groups always suffer from a higher packet loss rate comparing to consistent groups. The result is reasonable because frequent peer join and peer leave mean that the server has to spend more time on group management. In addition, the server needs to send new RESPONSIBILITY_MAP to the affected clients as well. It also costs some time for clients to set up WebRTC connection with new groupmates. On account of the 10% packet loss rate threshold, the server can support 32 normal peers even if groups change frequently.

4.5 Slow Peer Capacity Test

Figure 4–5 shows server's capacity when all peers are slow peers. If all the peers are slow peers, there is no peer-to-peer communication anymore and only client-server data transfer exists. Considering that data channel employed between client and server is a reliable data channel, packet loss rate is no longer a suitable benchmark to weigh the server's capacity. Instead, we use frame rate to measure it. From Figure 4–5 we can see that frame rate drops when peer number rises. The

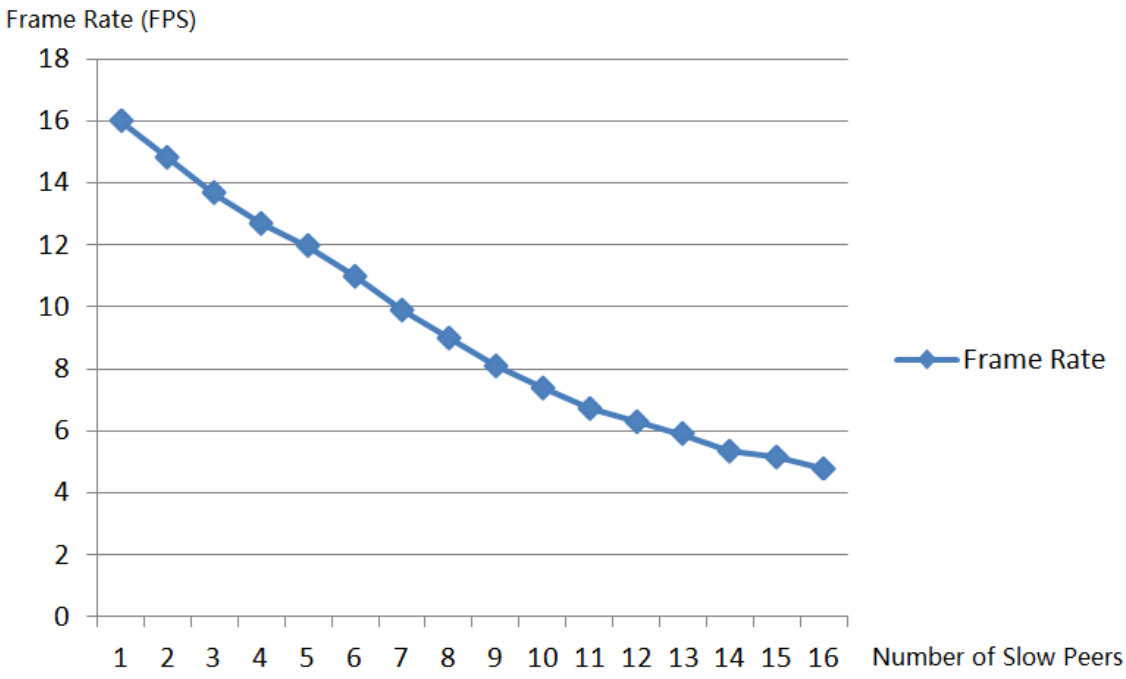

Figure 4–5: Frame rate changes with the number of slow peers

reason is quite straightforward. An increasing number of slow peers means more transmission time and bandwidth consumption, which can make the server congested and overwhelmed. If 10 frames per second is the minimum frame rate allowed, then the server can support no more than 6 slow peers.

Recall that there is no P2P communication among the clients when all the clients are slow. At this time, the server works exactly the same as a server using traditional client-server transmission mode. So the server's capacity for slow peers equals its capacity when our P2P algorithm is not applied. The server's capacity is 6 when using traditional client-server transmission mode and it is increased to 32 by our algorithm.

4.6 Summary

ImageHash Function Performance Test - According to the experiments on the imageHash function, it can be seen that imageHash does not work as we expected. When the server applies the imageHash function, clients suffer from a more severe packet loss rate. The main reason is that the imageHash function is resource-consuming and the bottleneck for the server at this moment is processing time instead of bandwidth. The performance of the server depends greatly on the efficiency of the imageHash function.

Group Size Test - When testing different size of the groups, we can see that growing the size of the groups increases the clients' burden to forward the packets received from the server. Finally, clients are overwhelmed by frequent data exchange within their groups. On the other hand, it cost the server more bandwidth and time to transmit data to the clients when the groups are small. So we need to find a

balance point for the group size. Based on the experiments, 8 is found as the default group size.

Group Consistency Test - When considering the dynamic nature of Internet, where peers join and leave the groups frequently, the server needs more time to manage the groups and the clients suffer from a larger packet loss rate. Therefore, the capacity of the server in the real world is less than that in the ideal condition.

Slow Peer Capacity Test - Depending on the experiments, the server can only support at most 6 slow peers, if it has to provide every peers with full frames. In other words, the server can only support 6 clients if the P2P data transmission mechanism is not applied. Considering that its capacity becomes 32 when using our P2P algorithm, we can draw the conclusion that the algorithm improves the system's capacity to a large extent.

CHAPTER 5
Conclusions and Future Work

5.1 Conclusions

Traditional P2P protocols rely on either client-end software or browser plug-ins to coordinate data exchange between clients. Furthermore, no existing P2P protocols can achieve real-time video distribution. These two drawbacks prevent P2P protocols from being widely deployed in the field of real-time video distribution even though they save considerable bandwidth for servers.

In this work, we design a hybrid system with both the real-time feature of client-server transmission mode and the scalability feature of P2P protocols. Compared with the traditional client-server transmission mode, our server can support more users. According to the experiments discussed in the last chapter, the server can only support 6 clients when completely using client-server transmission; however, when P2P data exchange is enabled, 32 peers can enjoy video service from the server. Although our server is not a fully scalable server that can support hundreds of clients, its capacity is still greatly increased.

Our server also addresses the second drawback by achieving low-latency compared to other P2P protocols. The frames displayed on clients' screens are at most 4 seconds behind the live video while traditional P2P protocols usually have a delay of dozens of seconds.

Finally, by applying WebRTC to implement peer-to-peer data exchange, no client end software or browser plug-ins need to be installed. All clients need to do is to open a web page and the server will help coordinate WebRTC connection between peers. Based on the experiments, we can see that if neither the server nor clients are overwhelmed, packet loss rate is less than 1% even if the unreliable data channel is deployed. Thus, we can draw the conclusion that WebRTC can be the data channel to support a P2P protocol.

5.2 Future Work

Although our work achieves real-time, scalability and is plug-in free, we can also find several limitations of WPRS from experiments. More work about the imageHash function, server capacity, slow peer handling mechanism, grouping algorithm and video encoding still needs to be done in the future to improve the system.

5.2.1 imageHash

The most significant problem found from the experiments is the imageHash function overloading the server. Although imageHash can indeed prevent similar pieces from being sent again and reduce the server's total transmission time, it fades next to its negative effect.

We can use a fast machine to improve the server's performance and capacity. But if we want to completely free the server from the imageHash function, we need an extra machine focusing on the imageHash function, as shown in 5–1. As soon as the frame is captured, it is first received by the imageHash processor. The image-Hash processor forwards the frame to the main server immediately and then execute imageHash. After imageHash is finished, the hash values are sent to the main server.

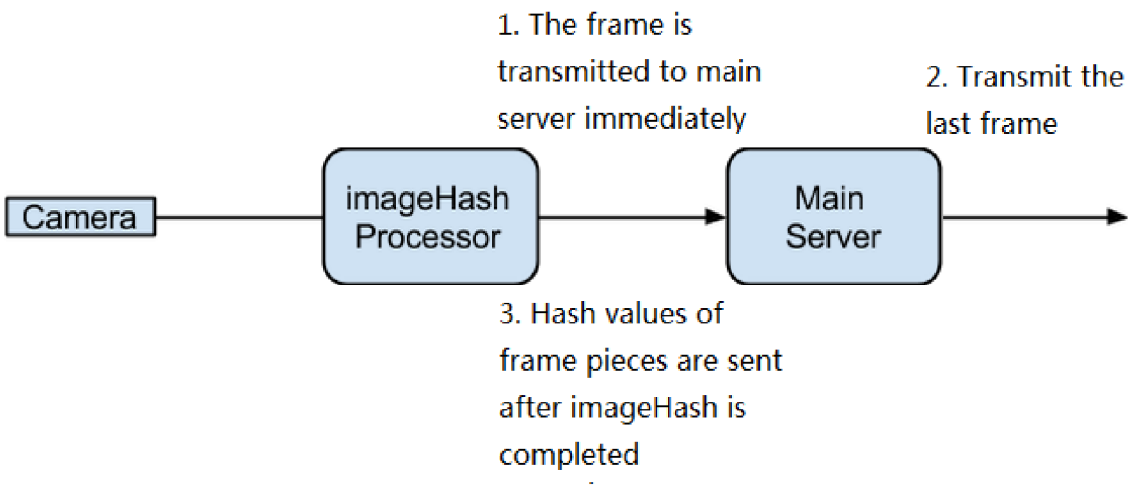

Figure 5–1: An extra imageHash processor is added to focus on imageHash

Therefore, imageHash and data transmission can be done simultaneously and do not affect each other.

It is important to insure that imageHash processor's message frequency is properly controlled. If it processes faster than the main server's transmission frequency, the main server may be overwhelmed by incoming frames and imageHash values. At the same time, if it takes too long to do imageHash, then the main server must wait for imageHash values before transmitting pieces to peers in time.

5.2.2 Server Capacity

Although the server's capacity is greatly improved by applying P2P feature, it is still not enough for a server. The main reason is that the server has to finish a complete process of pretreatment, group management and transmission before the next frame is captured. With an increasing number of clients, the transmission time

will also inevitably rise and overload the server. Besides, since images are sent instead of video, data can not be fully compressed to decrease transmission time.

In future work, I think the frame distribution algorithm should be improved to reduce transmission time so that the server can have a larger capacity. Based on the current mechanism, every frame is cut into smaller pieces and sent to each peer. Even though this mechanism minimizes redundant frame data sent, it also raises the number of packets sent between clients and the server. According to the experiments discussed in the last chapter, the server's current bottleneck is processing time instead of bandwidth. Thus, it may not be an efficient mechanism to cut every frame to pieces. If we consider the frames as the smallest data unit and distribute different frames to different peers, the number of packets sent can be greatly cut down and transmission time can also be reduced, with the sacrifice of more bandwidth usage.

5.2.3 Slow Peer Handling

Based upon the experiments, the server can support at most 6 slow peers or 32 normal peers. When the server determines to provide a slow peer full frames, the server's capacity is decreased by 6 or 7. Even if slow peers only receive frames with a lower resolution and frame rate, they still cost a bit deal of transmission time and cannot contribute.

As a result, slow peer management mechanisms should also be reformed in future work. One possible solution to set up a 1-1 relation between normal peers and slow peers. Every normal peer is responsible for a corresponding slow peer and sends the slow peer frames with a lower resolution and frame rate. As a normal peer can collect the whole frame first and then send it to the slow peer in one packet, the

transmission time for the normal peer should be acceptable. The number of slow peers allowed equals the number of normal peers at most and the server does not need to burden these slow peers at all.

5.2.4 Grouping Algorithm

Currently, all the peers are grouped randomly. This is not a problem right now because, in these experiments, all the machines used are located in the same laboratory so that delay among these machines can be ignored. However, when the server is put into use in the real world, the fact that group members are located all over the world will be an issue. As a result, a grouping algorithm is necessary to put peers close to each other into the same group.

5.2.5 Video Encoding

In order to use the imageHash function, we choose to send images instead of video streams. However, as the imageHash function fails to improve the server's performance, it maybe a better idea to distribute video streams to clients. First, the server cut the video stream captured in the same way as we cut frames. Then the server has several video streams of smaller size and distribute them to clients. The clients forward the video stream to their groupmates and also collect video steams from the group. At last the clients splice these video streams and display them on the screen.

This idea sounds to be feasible, but more researches are needed to achieve it. For example, it is a problem for the clients to forward the video stream received from the server to their groupmates efficiently. Furthermore, the clients need an algorithm to synchronize all the received video streams so that they can be played together.

References

[1] Sandvine, "Global internet phenomena report 2h 2014." https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/2h-2014-global-internet-phenomena-report.pdf, 2015.

[2] H. Tsukayama, "Youtube: The future of entertainment is on the web." Published on Washington Post. http://www.washingtonpost.com/business/technology/youtube-the-future-of-entertainment-is-on-the-web/2012/01/12/gIQADpdBuP_story.html, January 2012.

[3] L. L. Fernández, M. P. Díaz, R. B. Mejías, F. J. López, and J. A. Santos, "Catalysing the success of webrtc for the provision of advanced multimedia real-time communication services," in *Intelligence in Next Generation Networks*, IEEE, October 2014.

[4] S. Dutton, "Getting started with webrtc." http://www.html5rocks.com/en/tutorials/webrtc/basics/, 2012.

[5] M. Maheswaran and D. Bhattacharya, "Reality over web: Pervasive computing meets the web," *Tsinghua Science and Technology*, vol. 18, pp. 568 – 576, December 2013.

[6] F. Huang, *On Reducing Delays in P2P Live Streaming Systems*. PhD thesis, Virginia Polytechnic Institute and State University, Oct. 2009.

[7] I. Fette and A. Melnikov, "Can p2p real-time streaming video be successful?." Quora Answer https://www.quora.com/Can-P2P-real-time-streaming-video-be-successful, 2010.

[8] S. E. Deering, "Host extensions for ip multicasting." IETF Standard. http://tools.ietf.org/html/rfc988, 1986.

[9] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen, "Deployment issues for the ip multicast services and architecture," *Network*, vol. 14, pp. 78 – 88, Jan/Feb 2013.

[10] Y. hua Chu, S. G. Rao, S. Seshan, and H. Zhang, "A case for end system multicast," *Selected Areas in Communications*, vol. 20, pp. 1456 – 1471, October 2002.

[11] E. W. Biersack, P. Rodriguez, and P. Felber, "Performance analysis of peer-to-peer networks for file distribution," in *QoFIS*, Springer Berlin Heidelberg, October 2004.

[12] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai, "Distributing streaming media content using cooperative networking," in *NOSSDAV '02*, Association for Computing Machinery, Inc., May 2002.

[13] S. Krause and C. Hubsch, "Scalable application layer multicast," in *Consumer Communications and Networking Conference*, IEEE, January 2010.

[14] D. A. Tran, K. A. Hua, and T. Do, "Zigzag: an efficient peer-to-peer scheme for media streaming," in *INFOCOM 2003*, IEEE, March/May 2003.

[15] F. Rhinow, P. P. Veloso, C. Puyelo, S. Barrett, and E. O. Nuallain, "P2p live video streaming in webrtc," in *WCCAIS*, pp. 1 – 6, IEEE, January 2014.

[16] D. Xu, M. Hefeeda, and S. Hambrusch, "On peer-to-peer media streaming," in *ICDCS'02*, pp. 363 – 371, IEEE, 2002.

[17] S. Jin and A. Bestavros, "Cache-and-relay streaming media delivery for asynchronous clients," in *the 4th International Workshop on Networked Group Communication*, October 2002.

[18] Y. Cui, B. Li, and K. Nahrstedt, "ostream: asynchronous streaming multicast in application-layer overlay networks," *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 91 – 106, January 2004.

[19] Y. Cui and K. Nahrstedt, "Layered peer-to-peer streaming," in *NOSSDAV '03*, pp. 162–171, June 2003.

[20] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava, "Promise: peer-to-peer media streaming using collectcast," in *ACM Multimedia (MM'03)*, pp. 45–54, November 2003.

[21] Y. Guo, K. Suh, J. Kurose, and D. Towsley, "P2cast: peer-to-peer patching scheme for vod service," in *WWW'03*, pp. 301–309, May 2003.

[22] H. Deshpande, M. Bawa, and H. Garcia-Molina, "Streaming live media over a peer-to-peer network," technical report, Stanford InfoLab, April 2001.

[23] P. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, "From epidemics to distributed computing," *Computer*, vol. 37, pp. 60 – 67, May 2004.

[24] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM Transactions on Computer Systems*, vol. 17, pp. 41 – 88, 1998.

[25] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming," in *INFOCOM 2005*, pp. 2102 – 2111, IEEE, March 2005.

[26] B. Li, S. Xie, Y. Qu, G. Y. Keung, C. Lin, J. Liu, and X. Zhang, "Inside the new coolstreaming: Principles, measurements and performance implications," in *INFOCOM 2008*, pp. 13 – 18, IEEE, April 2008.

[27] A.-M. K. Ayalvadi J. Ganesh and L. Massoulié, "Peer-to-peer membership management for gossip-based protocols," *IEEE Transactions on Computers*, vol. 52, pp. 139 – 149, February 2003.

[28] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan, "Webrtc 1.0: Real-time communication between browsers." W3C Editor's Draft `http://w3c.github.io/webrtc-pc`, 2015.

[29] B. Leiba, A. Cooper, and B. Campbell, "Rtcweb status pages." IETF Draft `http://tools.ietf.org/wg/rtcweb/charters`, 2015.

[30] G. A. Eriksson and S. Håkansson, "Webrtc: enhancing the web with real-time communication capabilities." Published on Ericsson Review `http://www.ericsson.com/cn/news/120405_webrtc_enhancing_the_web_with_real-time_communication_capabilities`, 2012.

[31] S. Loreto and S. P. Romano, "Real-time communications in the web: Issues, achievements, and ongoing standardization efforts," *Internet Computing*, vol. 16, pp. 68 – 73, Sept/Oct 2012.

[32] J. K. Nurminen, A. J. R. Meyn, E. Jalonen, Y. Raivio, and R. G. Marrero, "P2p media streaming with html5 and webrtc," in *INFOCOM WKSHPS*, pp. 63 – 64, IEEE, April 2013.

[33] E. Klinger and D. Starkweather, "phash - the open source perceptual hash library." http://www.phash.org/, 2008.

[34] C. Zauner, "Implementation and Benchmarking of Perceptual Image Hash Functions," Master's thesis, Upper Austria University of Applied Sciences, Hagenberg Campus, July 2010.

[35] I. Fette and A. Melnikov, "The websocket protocol." IETF Standard. https://tools.ietf.org/html/rfc6455, 2011.