Chapter 1

INTRODUCTION

1.1 Overview

The popularity of video streaming systems has grown tremendously in the past few years. Sites like YouTube1 support user generated video content and contribute a significant amount of Internet traffic The latest market research conducted by Cisco company indicates that video streaming accounts for 53% of the mobile Internet traffic in 2013 and will reach 69% by the year 2018. In parallel, global mobile data is expected to increase 11-fold in the next five years and is expected to contribute about 50% of Internet traffic by 2012. The quality of video streaming is largely dependent on the downlink Internet bandwidth available to the end user. Although Web sites like YouTube support high quality videos, due to the limited Internet bandwidth available today, many users stream videos at low quality and compromise on their user experience. Some systems dynamically adapt the quality of videos based on the available bandwidth, but still end up streaming videos at low quality because of the limited bandwidth. Additionally, video streaming in mobile devices, like smart phones, are becoming increasingly important for video delivery. Unfortunately, Internet bandwidth to mobile devices is expected to take a long time to meet the demand for real-time streaming. Hence, despite the popularity, providing bandwidth for high-quality Internet video streaming still remains a challenge in many scenarios.

1.2 Problem statement

Despite the rapid advancements in network infrastructures, it is still challenging to deliver high-quality streaming video over wireless platforms. On one hand, the Wi-Fi networks are limited in radio coverage and mobility support for individual users; On the other hand, the cellular networks can well sustain the user mobility but their bandwidth is often inadequate to support the throughput-demanding video applications. The increase in internet traffic quantity over mobile phones and tablets has put more emphasis on technologies such as 4G to provide a larger bandwidth for streaming. To overcome the problem of limited bandwidth for streaming, we propose Concurrent Multipath Transfer (CMT) of video frames. CMT is a system that aggregates bandwidth from multiple cooperating users in a neighbourhood.

1.3 Objectives

The primary objective of CMT is to efficiently and proportionately divide the video frames amongst the neighbours and the client system to effectively enhance the video streaming speed across heterogeneous networks.

1.4 Motivation

The motivation for the system stems from the fact that although individual users may have limited bandwidth, the high density of Internet users in a neighbourhood with only a fraction active at any given time provides the opportunity to exploit the unused bandwidth to improve video streaming quality. When a user streams a video, CMT aggregates bandwidth by connecting to nearby cooperating users and uses their Internet connections in addition to the user's own connection. As the number of neighbours contributing to the client bandwidth increases, CMT can stream videos at higher quality. In the current implementation, CMT does not use enhancement layers but streams individual frames through multiple links for better throughput and video quality. CMT exploits the fact that devices with multiple network interfaces can connect to other devices simultaneously while being connected to the Internet. CMT exploits the fact that devices with multiple network interfaces can connect to other devices simultaneously while being connected to the Internet.

Chapter 2

LITERATURE SURVEY

[2.1] "COMBINE: Leveraging the Power of Wireless Peers through Collaborative Downloading" - G. Ananthanarayanan, V. Padmanabhan, L. Ravindranath and C. Thekkath, In Proceedings of ACM Mobisys, San Juan, Puerto Rico, June 2007.

COMBINE is a system for collaborative downloading that uses both the WLAN and the WWAN in combination, in an attempt to bridge the range-speed dichotomy. Nodes in close vicinity use the high-speed WLAN to discover each other, form a collaboration group, and stripe traffic across their WWAN links, increasing the effective WAN download speed available to any one node. All of these steps happen automatically, with user involvement limited to setting policy. COMBINE aggregates bandwidth from multiple nearby phones by forming a wireless ad-hoc network between the nodes. The 3G bandwidth on the phones is then combined to improve download times over HTTP. The main focus of COMBINE is an incentive system that is based on battery energy cost, with an accounting system that pays and bills users based on their bandwidth contribution. COMBINE supports only file transfer, it does not have specific constraints on the upload or the download times, and hence can use straight-forward work distribution techniques to assign a flow across multiple links. CMT supports real-time video streaming and distributes frames across multiple connections for improved video quality. CMT effectively utilizes the available bandwidth and dynamically adapts to neighbours joining and leaving the network.

[2.2] "Improvements on Block Size Control Method for Adaptive Parallel Downloading", J. Funasaka, K. Nagayasu, and K. Ishida, In Proceedings of the International Workshops on Distributed Computing Systems, pages 648 – 653, Washington D.C., USA, March 2004.

This paper proposes a system to fetch different parts of a file from the servers in parallel from the point of view of downloading time and fault-tolerance of network service rather than use the traditional downloading programs that usually select one server and retrieves a file from that server. The paper proposes an adaptive parallel downloading method, which enables us to stabilize and accelerate the downloading time even in the network environments with various bandwidths. The new method has the following features: independence of the number of mirror servers and introduction of request pipelining into parallel downloading. This approach can be used to divide a complete file into smaller, fix-sized segments that can be retrieved. Based on practical experiments from the viewpoint of downloading speed and stability of downloading time, we confirm that the improved method is more effective than the previous methods. There is a possibility that the improved method become one of the most important network control technologies for network assurance.

[2.3] "Bandwidth Aggregation for Real Time Applications in Heterogeneous Wireless Networks", K. Chebrolu and R. Rao, IEEE Transactions on Mobile Computing, 5(4):388 – 403, April 2006.

This paper proposes a system that benefits from the simultaneous use of multiple interfaces and present a network layer architecture that enables diverse multi-access services. In particular, we explore in depth one such service provided by the architecture: Bandwidth Aggregation (BAG) for real-time applications. Bandwidth Aggregation assumes multiple Internet connections on the same device through multiple interfaces and aggregates bandwidth across these interfaces for video streaming. The main disadvantage here is that the number of network interfaces per device is limited (usually two) and the maximum capacity this system can achieve is the sum of the Internet bandwidths at these interfaces. CMT also does bandwidth aggregation but instead of combining bandwidth from network interfaces in a single device, CMT aggregates Internet bandwidth from multiple neighbours. Hence, CMT can achieve greater capacity than such multi-homed systems, only limited by the wireless capacity.

[2.4] "CMT-QA: Quality-Aware Adaptive Concurrent Multipath Data Transfer in Heterogeneous Wireless Networks," C. Xu, T. Liu, J. Guan, H. Zhang and G. M. Muntean, in *IEEE Transactions on Mobile Computing*, vol. 12, no. 11, pp. 2193-2205, Nov. 2013.

This paper proposes a novel Quality-aware Adaptive Concurrent Multipath Transfer solution (CMT-QA) which utilizes SCTP for FTP-like data transmission and real-time video delivery in wireless heterogeneous networks. CMT-QA monitors and analyses regularly each path's data handling capability and makes data delivery adaptation decisions in order to select the qualified paths for concurrent data transfer. CMT-QA includes a series of mechanisms to distribute data chunks over multiple paths intelligently and control the data traffic rate of each path independently. CMT-QA's goal is to mitigate the out-of-order data reception by reducing the reordering delay and unnecessary fast retransmissions. CMT-QA can effectively differentiate between different types of packet loss to avoid unreasonable congestion window adjustments for retransmissions. Although the path status is an important factor that affects the scheduling policy, the application requirements should also be considered to guarantee the QoS.

Chapter 3

SYSTEM ANALYSIS

3.1 Existing Systems

There has been considerable research on increasing Internet bandwidth and application throughput in recent years. One focus area is on effectively using the available bandwidth at a single network interface (for example, using download accelerators). With the proliferation of multi-homed devices, there has been an additional focus towards aggregating bandwidth from multiple Internet connections on a single device to improve application throughput. Recently, with the increasing popularity of devices with multiple interfaces such as smart phones, some research prototypes have focused on aggregating bandwidth across multiple devices to improve application throughput. Transport protocols beyond TCP and UDP can enable multi-link support, as well. Single file videos must be specially encoded to enable streaming over multiple links. This chapter describes some related work in these areas in detail and discusses how our work differs from each.

3.1.1 Download Accelerators

Download accelerators and peer-to-peer (P2P) systems improve file download speed by getting parts of the file over multiple connections. Download accelerators [RKB00] improve the download rate on a single Internet link by opening multiple connections to mirrored servers in parallel and downloading different parts of a file simultaneously. The download performance improves because a single bad server selection may severely impact the download performance, while downloading in parallel from multiple servers reduces the impact of a bad server selection.

In [RKB00], the authors implement a dynamic parallel-access scheme where clients connect to mirror sites using unicast TCP and dynamically request different pieces of a document from different sites, thus, adapting to changing network and server conditions. They built a prototype of the dynamic parallel access scheme as a JAVA client that takes the URL of the mirror servers as an input parameter. They evaluated their scheme with various mirrored sites for different document sizes under different network/server conditions. The results show dramatic speedups in downloading a document, even when network or server conditions change rapidly. When all

the servers used in the experiment have similar performance, then the speedup gain is very large. When the performances of the different servers are mismatched then the resulting speedup is not as significant when compared to the fastest server's performance. Even in this case, the parallel-access scheme achieves response times as low as the ones provided by the fastest server alone and at the same time eliminating the critical decision of server selection.

P2P networks can improve video stream rate by streaming different parts of a video from multiple nodes [LRLZ06]. In P2P networks, a client connects to multiple peers and parts of the file are downloaded from different peers. Similar to accessing multiple servers, P2P networks provide link diversity thereby improving the download performance.

[LRLZ06] reviews the state-of-the-art of peer-to-peer Internet video broadcast technologies. The authors describe the basic taxonomy of peer-to-peer broadcast and summarize the major issues associated with the design of broadcast overlays. They examine two approaches, namely, tree-based and data-driven, and discuss their fundamental trade-off and potential for large-scale deployment. In a 9 tree-based approach, peers are organized into structures (typically trees) for delivering data, with each data packet being disseminated using the same structure. Data-driven overlay designs contrast to tree-based designs in that they do not construct and maintain an explicit structure for delivering data. Instead they use the availability of data to guide the data flow.

Both download accelerators and P2P use multiple connections and parallel downloads to enhance their download performance. Although these systems can improve download speeds over traditional client-server systems, they can never achieve capacity more than the downlink bandwidth available at the end-host. In our project, we consider the scenario where the user Internet downlink is the bottleneck and increase its application bandwidth through multiple connections. We can achieve more than the user downlink bandwidth since the bandwidth of the neighbors is aggregated to improve video streaming. For our project the bandwidth gain is limited by the number of collaborating neighbors and not the client downlink capacity as in download accelerators and P2P systems.

3.1.2 Network Sharing

Network sharing is a well-explored field. Wireless Mesh Network (WMN) [AWW05] provides connectivity to users in a neighborhood which do not have direct Internet access. In mesh

networks, nodes in a neighborhood connect wirelessly to form a grid and share Internet access from one or a few nodes which do have an Internet connection. Wireless mesh networks consists of mesh routers, mesh clients and gateways. Mesh routers are dedicated nodes that operate in ad-hoc mode, usually stationary to support meshing. Gateways are nodes that have an Internet connection. Mesh clients can be stationary or mobile and communicate peer to peer with the mesh routers and gateways. There are three types of mesh 10 networks: infrastructure supported mesh networks, client wireless mesh networks and hybrid mesh networks. In infrastructure mesh networks, mesh routers provide the infrastructure for the clients. In client mesh networks, client nodes form a peer-to-peer network for extending Internet connectivity and perform actual routing. Hybrid meshing is a combination of infrastructure and client meshing. [AWW05] presents a detailed study on advances and challenges in wireless mesh networks. Wireless mesh networks are cost effective way of increasing Internet connectivity. They are self-organizing, self-healing and self-configuring. WMNs provide redundancy and involve multi-hop routing to transfer data to a gateway and back to a node. WMN protocols assign one gateway per flow and routes all the packets in a flow through the same path. WMN has scalability issues and suffers in performance as many nodes join the mesh network. Our project is similar in theme to mesh networks in forming a neighborhood network, but unlike mesh networks which uses a single Internet connection per flow, our nodes aggregate bandwidth from all nearby nodes in addition to its own Internet connection. It also splits a single video stream across multiple Internet connections.

3.1.3 Bandwidth Aggregation

There have been several research efforts recently on aggregating Internet bandwidth from multiple connections. Bandwidth aggregation techniques seamlessly use multiple Internet connections as if it is a fat connection. The system presented by Chebrolu [CR06] assumes multiple Internet connections on the same device through multiple interfaces and aggregates bandwidth across these interfaces for video streaming. An important aspect of an architecture that does bandwidth aggregation for real-time applications is the scheduling algorithm that partitions the traffic onto different interfaces such that the QoS requirements 11 of the application are met. [CR06] proposes Earliest Delivery Path First (EDPF), an algorithm that ensures packets meet their playback deadlines by scheduling packets based on the estimated delivery time of the packets. They show through analysis that EDPF performs close to an

idealized Aggregated Single Link discipline, where the multiple interfaces are replaced by a single interface with the same aggregated bandwidth. Using a prototype implementation and simulations carried using video traces, they show performance improvement with EDPF scheduling over using just the highest bandwidth interface and other scheduling approaches based on weighted round robin.

The number of network interfaces per device is limited (usually two) and the maximum capacity these systems can achieve is the sum of the Internet bandwidths at these interfaces. Our project also does bandwidth aggregation but instead of combining bandwidth from network interfaces in a single device, it aggregates Internet bandwidth from multiple neighbors. Hence, it can achieve greater capacity than such multi-homed systems, only limited by the wireless capacity (up to 600 Mbps in IEEE 802.11n [80211N]).

Systems discussed thus far attempt to improve throughput by parallel access to mirrored servers or simultaneous usage of multiple network interfaces. Recent research has focused on exploiting bandwidth at nearby nodes to improve application performance. COMBINE [APRT07] is a research prototype which aggregates bandwidth from multiple nearby phones by forming a wireless ad-hoc network between the nodes. COMBINE aggregates the 3G bandwidth on phones to download large files using HTTP. It improves HTTP download by getting different chunks of a file through different links. COMBINE uses an adaptive workload distribution algorithm to farm out work across the participants in the collaboration group. COMBINE uses HTTP byte-range requests for parallel 12 downloads and hence does not require server support. The main focus of COMBINE is an incentive system that is based on battery energy cost. An accounting system pays and bills users based on their bandwidth contribution. COMBINE includes an energy-efficient protocol for nodes to discover each other, exchange their bids, and form a collaboration group. The authors have prototyped COMBINE on Windows XP and evaluated its performance on laptop-class devices equipped with 802.11b WLAN NICs and GPRS WWAN modems. They show near-linear speedups for group sizes of up to five nodes

Our project is similar to COMBINE in forming an ad-hoc network with neighbor nodes and aggregating bandwidth from multiple neighbors. It applies bandwidth aggregation across multiple nodes to effectively improve video streaming performance. Video streaming is a high bandwidth application and unlike HTTP download, has real time constraints where the bandwidth gain from collaborating with neighbors can boost video performance tremendously.

Link-alike [JJKPS08] is similar to COMBINE but is used to improve the upload capacity of the client. Many photo, video sharing services and online backup services require users to upload large amounts of data to their Websites, but the asymmetric broadband connections prevalent in residential network pose a challenge to users who want to publish information. Link-alike tries to solve this problem by increasing the upstream capacity of the client by aggregating the uplink capacities of the nodes in neighborhood. Link-alike addresses the challenges of operating in an environment that is highly lossy, broadcast in nature and half-duplex. Link-alike uses opportunistic wireless reception, a novel wireless broadcast rate control scheme, and preferential use of the wired downlink. Through analytical and experimental evaluation, [JJKPS08] demonstrates that Link-alike provides significantly better throughput than previous solutions based 13 on TCP or UDP unicast. Our project also aggregates bandwidth in a neighborhood environment, but unlike Link-alike it is focused on aggregating downlink bandwidth for video streaming.

Since systems like COMBINE and Link-alike support only file transfer, they do not have any constraint on the upload or the download time, and hence use straight-forward work distribution techniques to assign a flow across multiple links. Our project supports real-time video streaming and distributes frames across multiple connections for improved video quality. It effectively utilizes the available bandwidth and dynamically adapts to neighbors joining and leaving.

3.1.4 Virtual Interfaces

Although our project assumes multiple interfaces at a device, it is not a requirement. Technologies like Multinet (Virtual WiFi) [CBB04] make a single wireless interface act as multiple network interfaces enabling them to connect to multiple nodes at the same time. For example, a node can connect to the Internet through an access point at the same time as it can connect to a neighboring node, both connections using the same wireless card. Multinet continuously switches a single wireless card across multiple networks. The system is transparent to the user and is agnostic to the upper layer protocols. Multinet is implemented as a Windows driver and virtualizes a single wireless card into multiple interfaces.

FatVAP [KLBK08] is a system which enables a single wireless card to connect to multiple access points at the same time using similar techniques used in Multinet. FatVAP is implemented as an 802.11 Linux driver that aggregates the bandwidth available at accessible

APs and also balances their loads. FatVAP chooses the APs that are worth connecting to and connects with each AP just 14 long enough to collect its available bandwidth. It ensures fast switching between APs without losing queued packets. FatVAP works with unmodified APs and is transparent to applications and the rest of the network stack. The authors evaluate FatVAP both in a lab, at hotspots and for residential deployments. FatVAP delivers a median throughput gain of 2.6x, and reduces the median response time by 2.8x.

Using approaches like Multinet or FatVAP, our project can be effective for devices with only one network interface. In our current implementation, devices have multiple interfaces but using technologies like Multinet and FatVAP it can work on devices with single network interface.

3.2 Proposed System and Architecture

3.2.1 Challenges

The Client and Neighbors should find each other and connect through an ad-hoc wireless network. Once connected, at any time the Client should have an updated knowledge of the active Neighbors willing to contribute bandwidth. Neighbors should be able to join and leave the system at any time during video streaming.

The Video Server should be able to stream a single video through multiple nodes. The server needs to know the Client and Neighbor end points to be able to send the frames. The server needs to distribute the frames to be sent based on the bandwidth available at each link and it should dynamically adapt to the changes in the bandwidth. An ideal distribution protocol will fully utilize the available bandwidth and proportionally distribute frames based on the ratios of the available bandwidths.

The system should adapt to a change in the network neighborhood. Neighbors can then join and leave at any time during the video streaming. When new neighbors join, the system should be able to adapt quickly and start using the newly available bandwidth. When a Neighbor leaves, the system should be adaptive and recover the lost frames and redistribute them across the remaining active links.

The Client should be able to buffer the frames received through different links and play them. The underlying mechanism of receiving frames from different links and playing them should

be opaque to the user. The system should have an appropriate buffering mechanism which decides whether to wait for the late frames or discard them.

3.2.2 Architecture

Our system design addresses all the above challenges. Figure 3.1 shows the architecture of the system. The system has three distinct components: the Video Server, the Client node which requests the video and the Neighbors which help to aggregate bandwidth for better video quality. It assumes that multiple interfaces are present in the Client and Neighbors so that they can form wireless ad-hoc network wireless communications while being connected to the Internet.

In brief, it works as follows. The Client forms a wireless ad-hoc network with Neighbors which are idle so that it can use their bandwidth. The Client periodically updates the Video Server with the Neighbor information and the server streams the video to the Client using both the Client and the Neighbor bandwidth. A participating Neighbor simply acts as a proxy in delivering the frames from the server to the Client. We explain the role of each component in greater detail below. We defer the implementation details to Chapter 6.

3.2.2.1 Client

Users request video using the Video Player in the Client. The Client then sends the request to the Video Server. The Video Server replies back with the meta-data of the video which consists of the number of frames and the frame rate. The Client forms an ad-hoc network with the Neighbors and informs the Video Server about the available links to stream the video. As the frames start arriving, they are buffered and later the video is played by the Video Player

The Client has four main components and the role of each component is explained in detail below.

a. Neighbor Manager

   The role of the Neighbor Manager is summarized below:

   1. When a user requests a video, the Neighbor Manager creates an ad-hoc network and waits for Neighbors to join the ad-hoc network.
   2. The Neighbor Manager periodically broadcasts REQUEST messages to find new Neighbors in the ad-hoc network which are willing to contribute bandwidth.
   3. The Neighbor Manager keeps an updated knowledge of active Neighbors willing to help in the system. The Neighbor Manager monitors for periodic heartbeat messages (I-CAN-HELP messages) from the Neighbors and constantly keeps track of neighbors

joining and leaving the neighborhood The Neighbor Manager periodically informs the Video Plan Manager about the active Neighbors and changes in neighborhood (new Neighbors joining and Neighbors leaving).

4. The Neighbor Manager receives video frames from the active Neighbors and forwards it to the Buffer Manager. It keeps track of the frames received from each of the Neighbors and informs the Video Plan Manager.

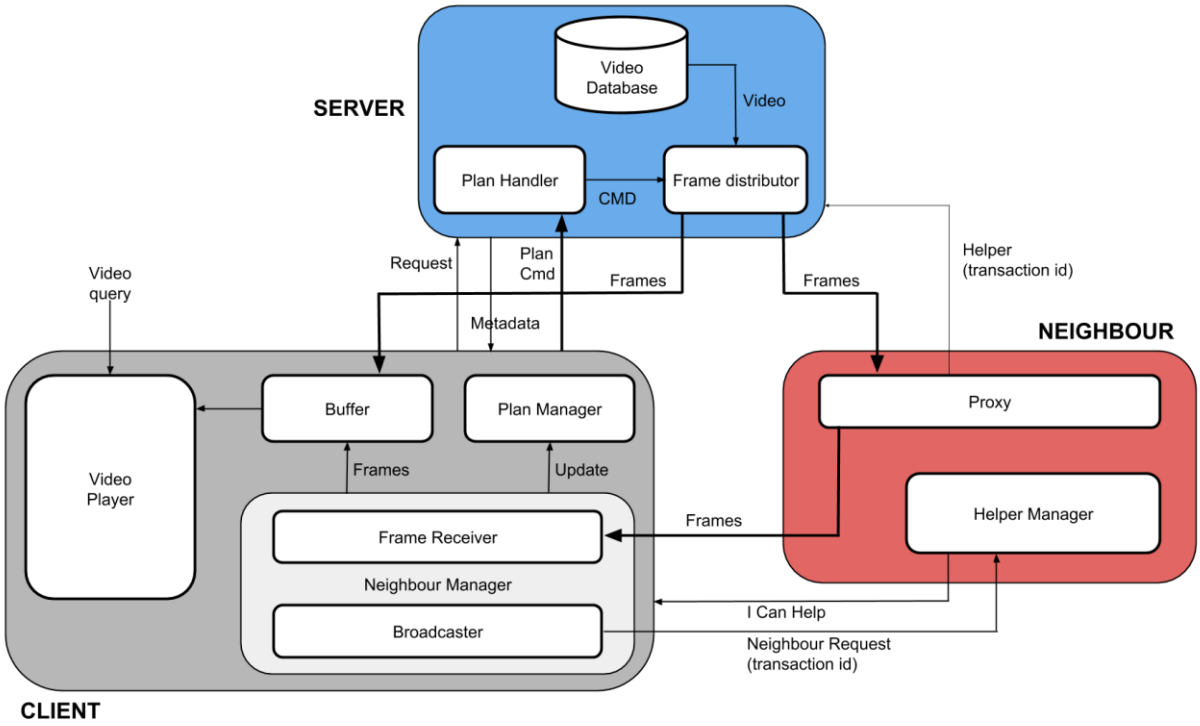

Figure 3.1: Proposed Architecture

b. Video Plan Manager

The Video Plan Manager is a core component of the Client that constantly informs the Video Server about the streaming plan. The role of the Video Plan Manager is summarized below.

1. The Video Plan Manager constructs the streaming plan with Neighbor details and periodically updates the Plan Handler in the Video Server. The streaming plan consists of information about the Client and Neighbor links (IP address and Port) that the Video Server can use to stream video to the Client.
2. When a new Neighbor, ready to contribute bandwidth, joins the ad-hoc network, the Video Plan Manager informs the Video Server about the Neighbor so that the Video Server can quickly use additional bandwidth to stream.

3. When a Neighbor leaves the network, the Video Plan Manager informs the Video Server so that the Video Server can recover lost frames and stream through other active links.

c. Buffer Manager

The Buffer Manager maintains the playout buffer that the Video Player uses to play the video.

1. Based on the meta-data response from the Video Server after the Client requests the video, the Buffer Manager initializes the playout buffer.
2. The Buffer Manager receives frames from the Video Server and stores them in the buffer.
3. The Buffer Manager also receives frames from the Neighbor through the Neighbor Manager and stores them in the buffer.

d. Video Player

The Video Player is the interface for the user in the topmost system.

1. A user can request a video using the Video Player in the Client.
2. The Video Player extracts frames from the playout buffer in the Buffer Manager and plays them.
3. The Video Player plays the video based on the meta-data of the video (frame rate).
4. When the frame to be played is missing, the Video Player stops playout and waits for late frames to arrive. The stop and buffering mechanism is explained in detail in the next chapter.

3.2.2.2 Neighbor

A Neighbor is idle and has spare bandwidth which it is willing to share with the Client to improve the video streaming quality. Though Figure 2 shows only one instance of the Neighbor, multiple Neighbors can contribute bandwidth to a single Client. The role of each of the Neighbor components is explained below.

a. Helper Manager

The role of the Helper Manager is summarized below:

1. When the Neighbor is willing to contribute bandwidth, it looks for an adhoc network created by the Client and joins the network.

2. When the Helper Manager receives a REQUEST message from the Client, it starts sending I-CAN-HELP messages periodically to the Client. In the ICAN-HELP messages, the Helper Manager puts the IP Address and the port which the Neighbor keeps open for the Video Server to stream the video.
3. When there is user activity or network activity (due to other applications), the Helper Manager stops sending I-CAN-HELP messages.
4. The Helper Manager disconnects from the ad-hoc network when the user exits the application.

b. Proxy

The Proxy component in the Neighbor functions as below:

1. The Proxy keeps a port open for the Video Server to stream video through it.
2. The Proxy receives frames from the Video Server and forwards it to the Client through the ad-hoc network. Neighbors do not buffer the received frames and immediately forward the frames to the Client.

3.2.2.3 Video Server

The Video Server stores the videos that Client can request. The Video Server streams a single video through multiple links. It adapts the streaming to changes in the network neighborhood. The role of each of the Video Server components is explained below.

a. Video Database

The Video Database stores the uploaded videos that Client nodes can stream:

1. The videos are encoded and stored in AVI format.
2. The Video Database splits a video into frames and allows queries to a specific frame in a video. The Frame Distributor extracts the frames from a video in the Video Database and distributes them through multiple links.

b. Plan Handler

The role of the Plan Handler is summarized below:

1. The Plan Handler receives a streaming plan from the Video Plan Manager in the Client. The Plan Handler initializes the Frame Distributor to stream according to the plan.
2. The Plan Handler receives plan updates from the Client about changes in the neighborhood. When the Client informs the Plan Handler about a new

Neighbor, it updates the Frame Distributor to include the new Neighbor in the streaming process. When the Client informs the Plan Handler about a Neighbor that left in the middle of streaming, it updates the Frame Distributor to stop sending frames via that Neighbor.

c. Frame Distributor

The Frame Distributor is the core component of the system that streams a single video through multiple links.

1. The Frame Distributor runs a frame assignment module that assigns frames in a video to stream through different links.
2. The Frame Distributor uses TCP to send the frames to Client and the Neighbor.
3. The Frame Distributor adapts to change in bandwidth and effectively utilizes the links. 4.
4. The Frame Distributor works with the Plan Handler to adapt to the changes in neighborhood. The Frame Distributor starts streaming to new neighbors when they join and recovers lost frames when a Neighbor leaves.

Chapter 4

REQUIREMENTS

4.1 Hardware Requirements

Our project as it a proof of concept, has some high requirements on hardware, which can be improved upon as described in Chapter 9. The detailed minimum hardware requirements for running the different modules are given below.

4.1.1 Client

a. Processor speed:
    1. Minimum: 2 GHz
    2. Recommended: 3 GHz
b. Memory
    1. Primary memory(RAM): 2 GB
    2. Secondary memory: 100 MB
c. Network components
    1. Wireless network adapter
    2. Ethernet interface
    3. WLAN interface

4.1.2 Server

a. Processor speed:
    1. Minimum: 2 GHz
    2. Recommended: 3 GHz
b. Memory:
    1. Primary memory(RAM): 2 GB
    2. Secondary memory: 50 MB
c. Network components:
    1. Wireless network adapter
    2. Ethernet interface
    3. WLAN interface

4.1.3 Neighbor

a. Processor speed:
    1. Minimum: 2 GHz
    2. Recommended: 3 GHz
b. Memory
    1. Primary memory(RAM): 1 GB
    2. Secondary memory: 50 MB
c. Network components
    1. Wireless network adapter
    2. Ethernet interface
    3. WLAN interface

4.2 Software requirements

The software requirements of our proposed system are described in detail below.

4.2.1 Functional requirements

The functional requirements of each module is described below.

a. Server
    1. The server must be able to receive requests from a client.
    2. The server must be able to distribute frames to multiple nodes simultaneously and must be able to adjust to changing number of neighbors.
b. Client
    1. The client must be able to receive the list of videos from the server and display the to the user.
    2. The client must be able create an ad-hoc connection with each neighbor that accepts the request broadcasted by the client.
    3. The client must be able to receive frame from the server and any participating neighbors, order them and play them back to the user.
    4. The client must able to identify and retrieve lost frames from the server.
c. Neighbor
    1. The neighbor must be able to receive requests for help from any clients and display a dialog.

2. The neighbor must be able to determine between a declined request and an accepted one.
3. The neighbor must be able to receive frames from the server and forward them to the client without much delay.

4.2.2 Non Functional Requirements

We establish non-functional requirements according to the parameters below.

a. Performance
    1. The server must respond to the client's request in a timely fashion, without any exasperating delays.
    2. The network between the client, server and the neighbor must not bottleneck the improvements gained by concurrent streaming of video packets.
    3. The video frames which are usually derived from a compressed video, occupy a large amount of space. This space must not be impractically large.
    4. The broadcast-receive/decline event, distribution of frames and the addition of the neighbors must be in real time, without noticeable delays.
b. Scalability
    1. The bandwidth aggregation must be extensible to all network interfaces currently available.
    2. The system must be able to handle a large number of neighbor connections.
    3. The server must be able to server multiple clients simultaneously, give the hardware needs.
c. Reliability
    1. The server must not fail when servicing clients.
    2. The video buffer must not stop when the network changes, frames are lost, or any other unexpected event. The required frame must be requested from the server and buffered.
d. Usability
    1. The system must provide user friendly interfaces.
    2. The video playback must be smooth, audio synchronized and not too slow.
    3. The neighbor's host must be able to use bandwidth for other tasks when not streaming frames for the client.

Chapter 5

SYSTEM DESIGN

5.1 Context Diagram

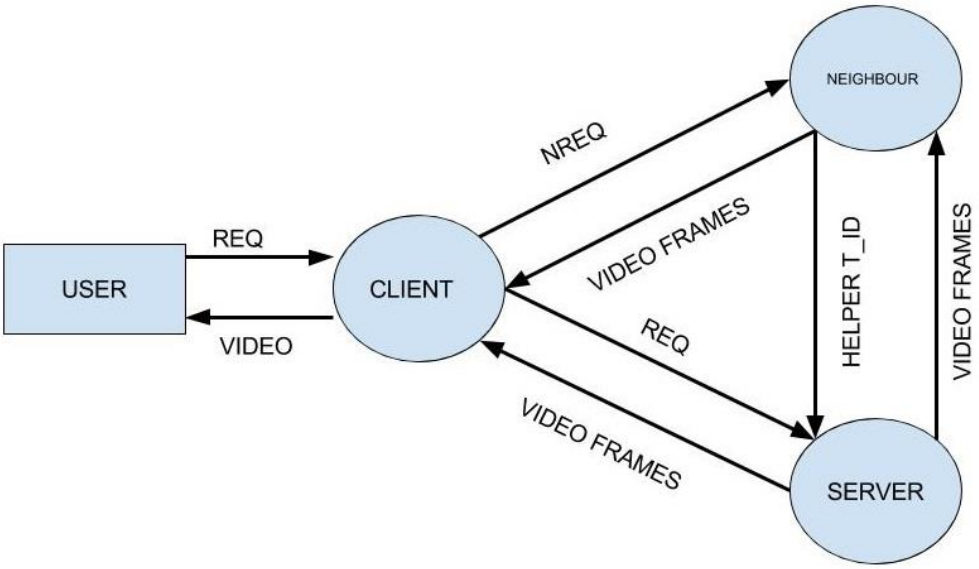

Figure 5.1: High level working

5.2 Data Flow Diagrams

Here we depict the data flow diagrams for the Server, Neighbour and Client

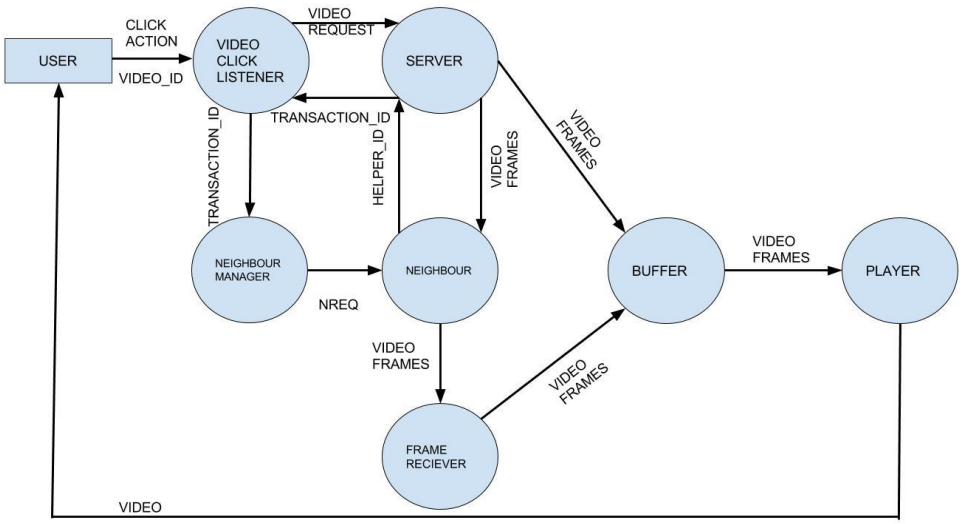

Figure 5.2: Level – 1 Server

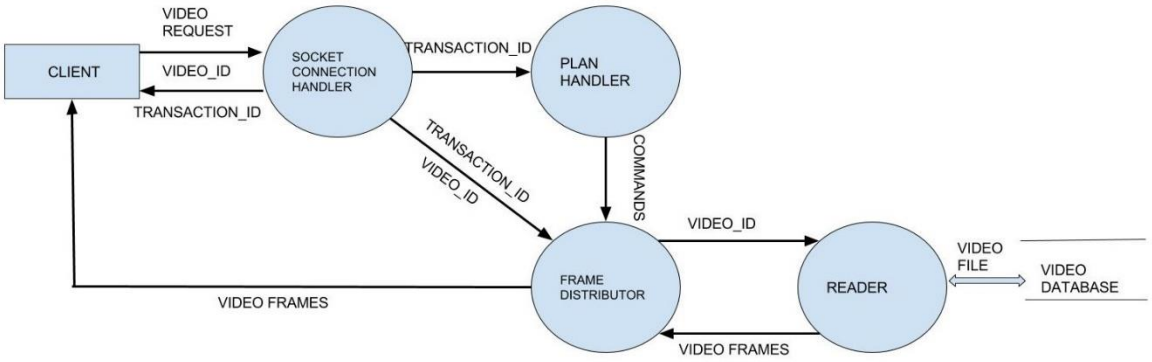

Figure 5.3: Level-1 Client

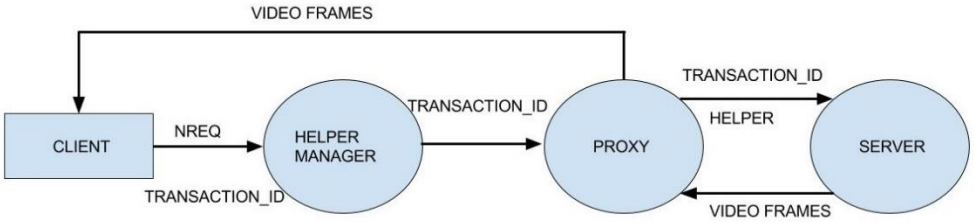

Figure 5.4: Level – 1 Neighbor

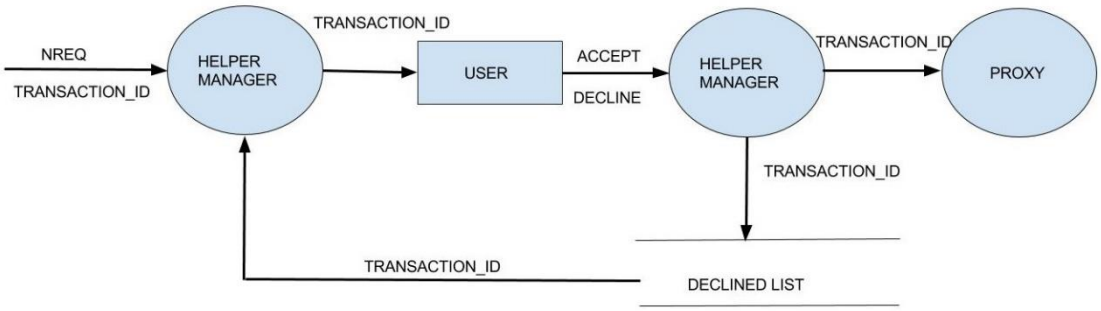

Figure 5.5: Level-2 Neighbor

Chapter 6

IMPLEMENTATION

6.1 Functional Implementation

This section presents implementation details of the protocols in the CStream system. Specifically, it discusses four areas: 1. Neighbor Management, 2. Frame Distribution, 3. Adapting to changes in neighborhood, 4. Buffering and Playing. We show illustrative examples to explain each of these areas.

6.1.1 Neighbour Management

This section explains the ad-hoc network formation and neighbor management. Specifically we explain the implementation details of Neighbor Manager in the Client and Helper Manager in the Neighbor and their interaction. An illustrative example to explain the sequence of flow in the implementation of CStream System is included.

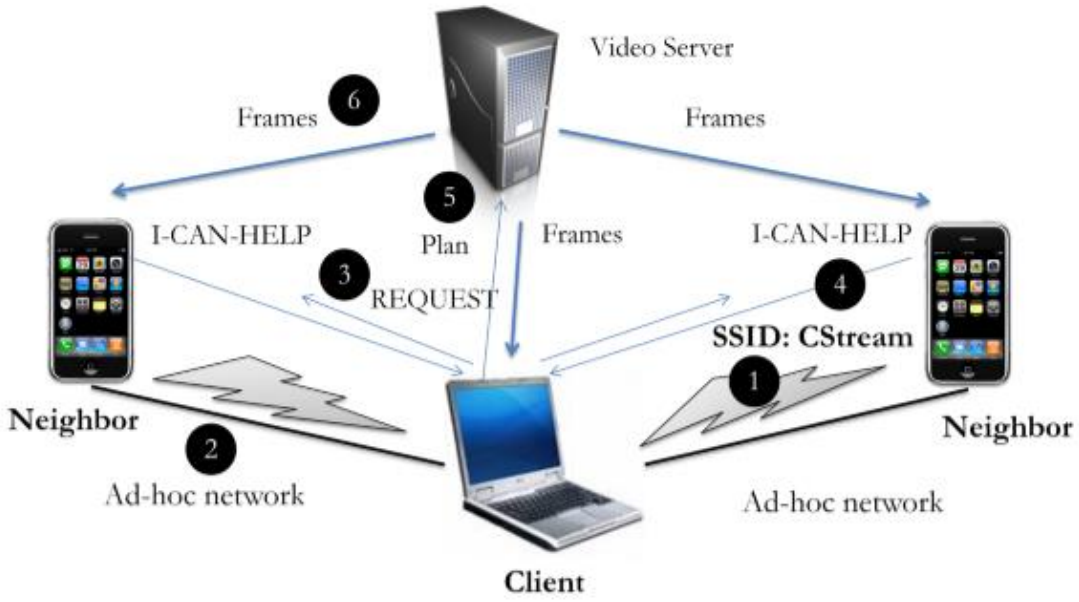

Figure 6.1: Ad-Hoc formation of Neighbor network

Figure 6.1 shows an example scenario with a Client and two Neighbors. Assume that a user requests a video using the Client. We describe the sequence of events step by step using Figure 6.1. In the figure, the steps are show in black circles with the step number.

Step 1: The Client creates an ad-hoc network with SSID CStream. If such a network already exists, the Client joins it. If the Client is already connected to the CStream ad-hoc network, it continues with Step 3.

The same ad-hoc network can be maintained across multiple video requests. For instance, the Neighbors may be connected in the ad-hoc network all the time even if a video is not requested. When it has user activity, the Neighbor disconnects from the ad-hoc network and connects back when it becomes idle again. This way of maintaining the ad-hoc network helps reduce the streaming start-up delay of forming the neighborhood network.

Step 2: Neighbors which are nearby the Client are in the range of CStream ad-hoc network. If they are running the CStream application and are idle, they join the network. Client and Neighbors get an IP address once they join the ad-hoc network.

Step 3: The Client broadcasts a REQUEST message in the ad-hoc network that it needs to stream a video and is looking for neighbors to contribute bandwidth. In our implementation, we use IP broadcast in the ad-hoc network to broadcast the request.

Step 4: When a Neighbor in the ad-hoc network receives a REQUEST message, it starts responding to the Client with I-CAN-HELP messages. The I-CAN-HELP messages contain the Neighbors IP address and port for the Video Server to stream the video via the Neighbor. The I-CAN-HELP messages are sent periodically to the Client (in our implementation, every second). The periodic ICAN-HELP messages from the Neighbors are used by the Client to determine if a Neighbor is still alive.

For each Neighbor, the table stores the IP address and the port for streaming, the last time when the Client received an I-CAN-HELP message from the Neighbor and the last received frame from the Neighbor. As detailed in subsequent sections, the Last Received Frame value will be used to recover lost frames when a Neighbor leaves. The Client decides that a Neighbor has left if it does not receive I-CAN-HELP messages for a specific amount of time (3 seconds in our implementation).

Step 5: The Client sends the streaming plan periodically (every 500 msec in our implementation) to the Video Server. The streaming plan consists of the list of end points (IP address and port) to stream. The Client sends its own end point and the end point of the Neighbors that are active (last update time is less than 3 seconds).The streaming plan changes

to capture the changes in the neighborhood (Neighbors joining and leaving) and the Client updates the Video Server with the new plan.

Step 6: Based on the streaming plan, the Video Server splits and streams the video across multiple links. The frame distribution protocol is explained in the section 6.1.2.

6.1.2 Frame Distribution

The Video Server sends a single video through multiple links. The video files are stored in AVI format in the Video Database. The Frame Distributor splits the video into frames and sends each frame through a single link. The sequence of flow at the server after the Client sends the initial streaming plan is illustrated with an example in Figure 6.2. The steps of the protocol are shown in black circles. In the example there are two active Neighbors and a Client.

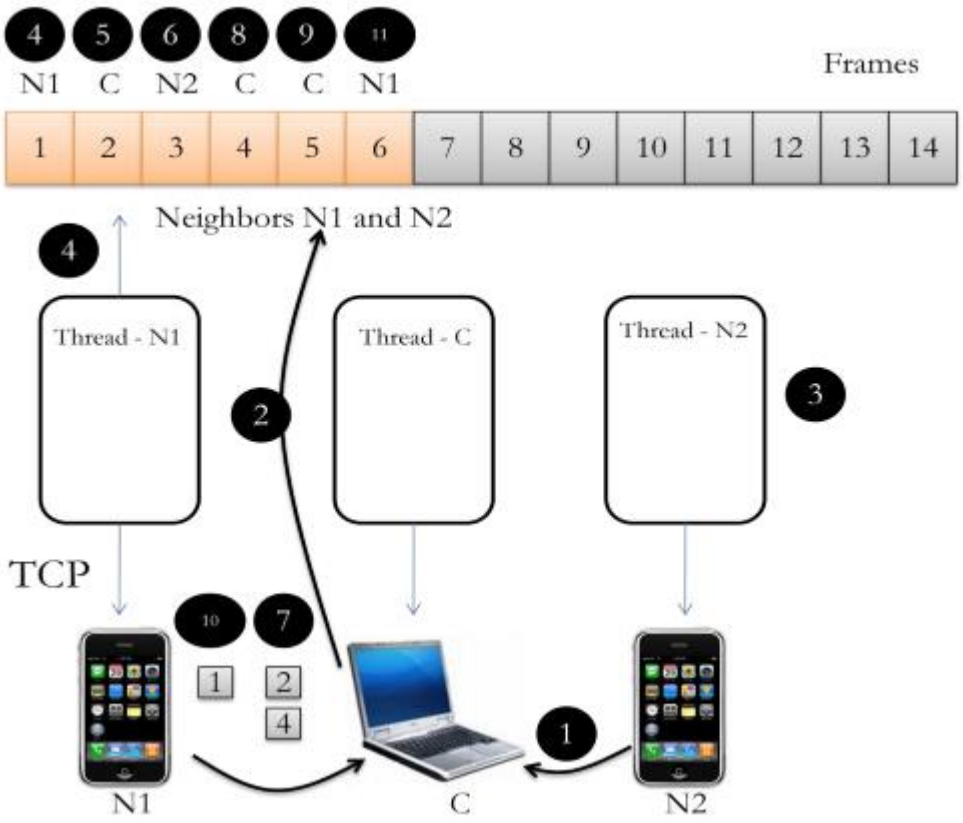

Figure 6.2: Distribution of frames

Step 1: As described in the section 6.1.1, the Neighbors periodically send I-CANHELP messages to the Client. Here Neighbor N1 and N2 sends periodic I-CAN-HELP messages.

Step 2: The Client informs the Video Server about the Neighbors N1 and N2 as part of the streaming plan.

Step 3: Now the server has three links for streaming video. The server creates three threads to simultaneously send frames through the three links. Each thread services one node. The job of each thread is to fetch frames and send it to the node.

The server keeps all the frames in a Frame Queue and runs a simple frame assignment process. Whenever a thread is ready to send a frame, it fetches the frame at the head of the Frame Queue and assigns the frame to that thread.

The frames are sent to the node using TCP. We rely on TCP to adapt and effectively utilize the available bandwidth in each link. When there is spare bandwidth, the thread requests the next frame from the Frame Queue and starts sending. In our implementation the TCP sender window is small, about 32 KB for each node and hence only one frame for our source videos is accommodated in the sender buffer before fetching the next frame. This eliminates the problem of multiple frames being queued on the sender side when the Client and the Neighbor link are slow.

Step 4: Thread N1 requests a frame and frame 1 that is at the head of the queue is assigned to the thread. The thread starts sending frame 1 to the Neighbor N1.

Step 5: Thread C requests a frame and is assigned frame 2.

Step 6: Thread N2 requests a frame and is assigned frame 3.

Step 7: Suppose the Client has three times more bandwidth compared to the neighbors. Frame 2 gets sent faster than frame 1 and frame 3.

Step 8: Since there is spare bandwidth in the Client link, thread C requests for another frame and is assigned frame 4.

Step 9: Frame 4 is also sent to the Client and the thread C requests another frame and is assigned frame 5.

Step 10: Thread N2 finish sending frame 1 to Neighbor N2, N2 finishes forwarding frame 1 to the Client.

Step 11: Now N2 requests another frame and is assigned frame 6. Thus, the system adapts to available bandwidth and proportionally distributes the frames to the links.

6.1.3 Adapting to changing Neighbourhood

Sections 6.1.1 and 6.1.2 explained the basic mechanism used for streaming. Often a neighborhood may change and, hence, the system should adapt to Neighbors joining and leaving. This section explains the mechanisms CStream uses to adapt to these changes.

6.1.3.1 Neighbour Joining

We continue to use the same example in the section 6.1.2 to explain how the CStream system adapts to new Neighbors joining the network.

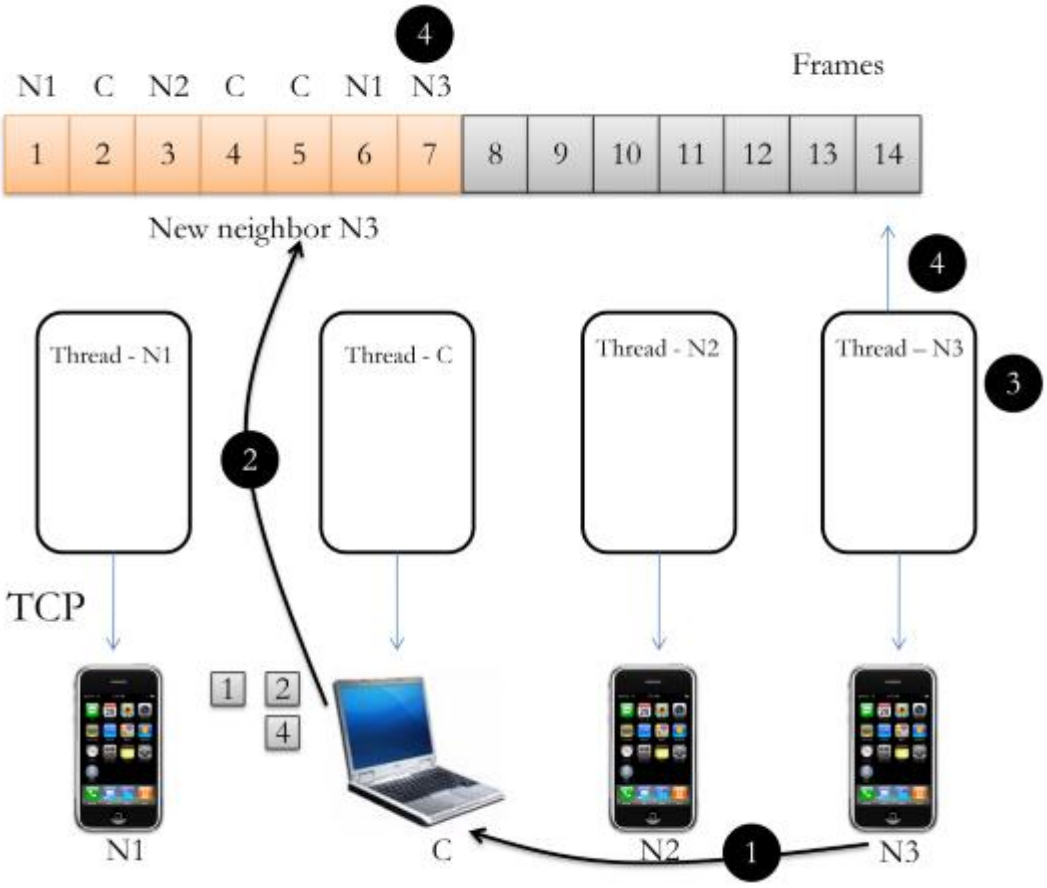

Figure 6.3: Neighbor joining

When a new Neighbor running CStream comes into the vicinity of the existing ad-hoc network (Neighbor N3 figure 5), the Neighbor joins the „CStream" ad-hoc network. It receives the periodic REQUEST broadcasts from the Client seeking help. The flow sequence of Neighbor joining scenario is explained step by step using Figure 6.3, a continuation of the example scenario shown in Figure 6.2.

Step 1: The Neighbor hearing the REQUEST messages starts responding to the Client with I-CAN-HELP messages.

Step 2: With a new entry in its Neighbor table, the Client updates the streaming plan, informing the Video Server about the new Neighbor.

Step 3: The server creates a new thread to send frames to the Neighbor N3.

Step 4: When a N3 thread requests a frame to send, it is assigned the next frame at the head of the queue (frame 7).

Thus, the CStream system adapts to new Neighbors joining and starts to use its bandwidth, quickly improving the overall throughput of the system.

6.1.4 Buffering and Playing

The Video Player in the Client plays the video as frames are being received. We implement a policy where the player stops and waits for late frames to arrive as opposed to discarding late frames. The Buffer Manager which receives frames from the Server and through the Neighbor stores them in the buffer for the Video Player to extract, decode and play. The buffering policy has the following features.

Initial Buffering: Before starting to play the video, the Video Player waits for some amount of frames to be initially buffered. In our implementation, the Video Player waits for first two seconds of the video to be buffered before starting to play. That is, we wait for (2 * fr) frames where fr is the frame rate. The choice of 2 to 4 seconds is common in many popular video players [oYT]. We found that around two seconds was optimal in our system both to decrease the startup delay and to give a good visual quality without too many rebuffer events. Stop and

Rebuffering: After the initial buffering, the player plays the frames continuously as long as they are in the buffer. The Player plays the video with the appropriate frame rate that was reported in the meta-data. When the frame to be played is not yet available in the buffer, the video player stops playing the video and triggers a rebuffer event. It waits for the frames to arrive.

Playing after Rebuffering: During a rebuffer event, the Player stops and waits for the next two seconds of frames (2 * fr) to be received before starting to play again. This is to reduce

the total number of rebuffer events. If the total number of frames in the video is less than (current frame + 2 * fr), it waits until all the frames are received before playing again.

To illustrate with an example, assume that the frame rate of a video is 15 and the total number of frames is 120. Before starting to play the video, the player waits for the first 30 frames to be received (2 seconds of frames). Once they are received it starts playing. Suppose the 45th frame is not available in the buffer when the player is supposed to play it. The Player stops and triggers a rebuffer event. The Player waits until all the frames up to the 75th frame (next two seconds worth of videos frames) is received before starting to play again. Suppose another rebuffer event happens at the 105th frame, the player waits until all the 120 frames are received before playing again.

6.2 Module-wise Implementation

The high level algorithms of each of the modules and their descriptive flow charts are given below. All the algorithms have been written in pseudocode.

6.1.1 Server

The server has two main components, the plan handler and the frame distributor. The high level working of the server is depicted in the flow chart below,, followed by its modules.

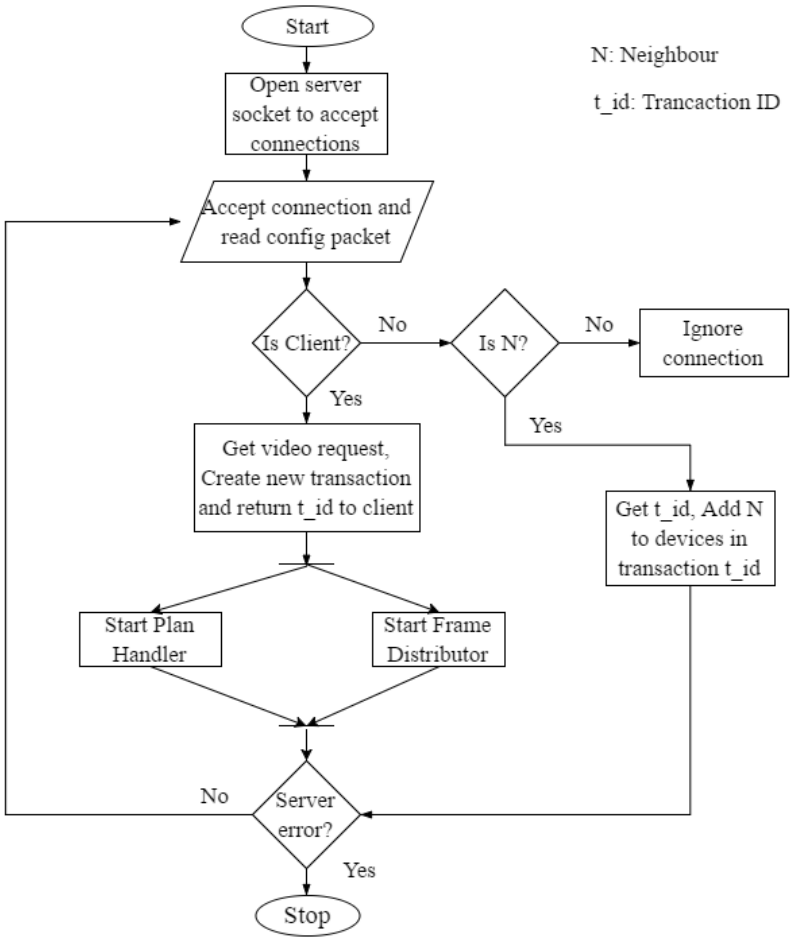

Figure 6.5: Flowchart of Server

a. Plan Handler:

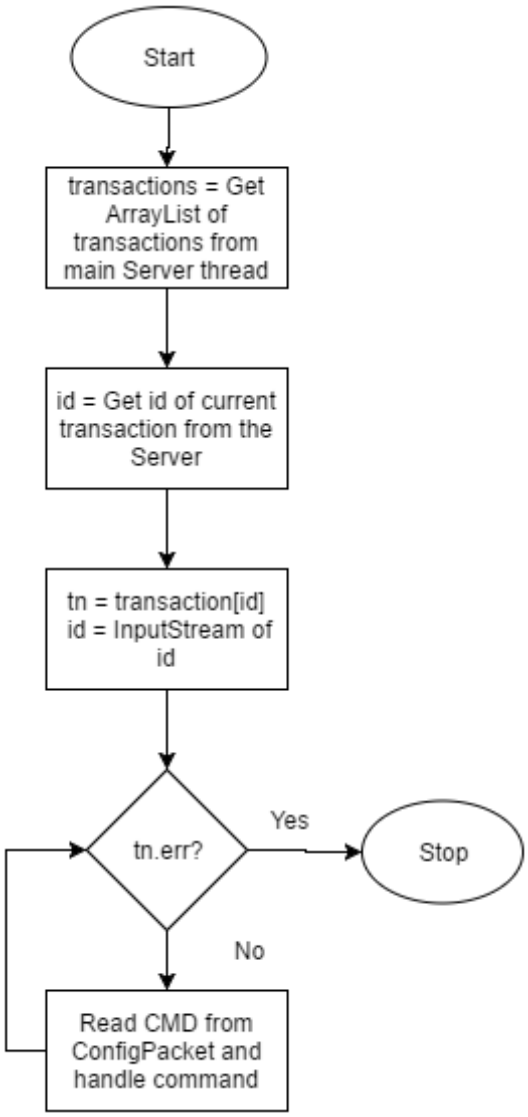

Figure 6.6: Flowchart of Plan Handler

The Plan Handler is implemented as follows

```
class PlanHandler implements Runnable {
    private Thread t;
    private String threadName;
    int tid;
    transaction tn;
    ObjectInputStream input;
    PlanHandler(int id){
        this.threadName = "Plan handler thread";
        tid = id;
        tn = (transaction)Metadata.getInstance().transactions.get(tid);
        input = (ObjectInputStream)tn.in;
    }
    @Override
```

```
    public void run() {
        for(;;) {
            if(tn.err)
                break;
            try {
                Object obj = input.readObject();
                if(obj instanceof configPacket) {
                    configPacket cp = (configPacket)obj;
                    if(cp.cmd == configPacket.CMD.STOP)
                        break;
                }
            } catch(IOException ex) {

Logger.getLogger(PlanHandler.class.getName()).log(Level.SEVERE, null, ex);
                break;
            } catch(ClassNotFoundException ex) {

Logger.getLogger(PlanHandler.class.getName()).log(Level.SEVERE, null, ex);
                break;
            }
        }
        tn.err = true;
        try {
            tn.in.close();
        } catch (IOException ex) {
            Logger.getLogger(PlanHandler.class.getName()).log(Level.SEVERE,
null, ex);
        }
        Iterator iter = tn.devices.iterator();
        while(iter.hasNext()) {
            Object device = iter.next();
            ObjectOutputStream out = (ObjectOutputStream)device;
            try {
                out.close();
                tn.devices.remove(device);
            } catch (IOException ex) {
                tn.devices.remove(device);

Logger.getLogger(PlanHandler.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
        System.out.println("Thread " +  threadName + " exiting.");
    }
    public void start() {
        System.out.println("Starting " +  threadName );
        if (t == null) {
            t = new Thread(this, threadName);
            t.start();
        }
    }
}
```

b. Frame Distributor:

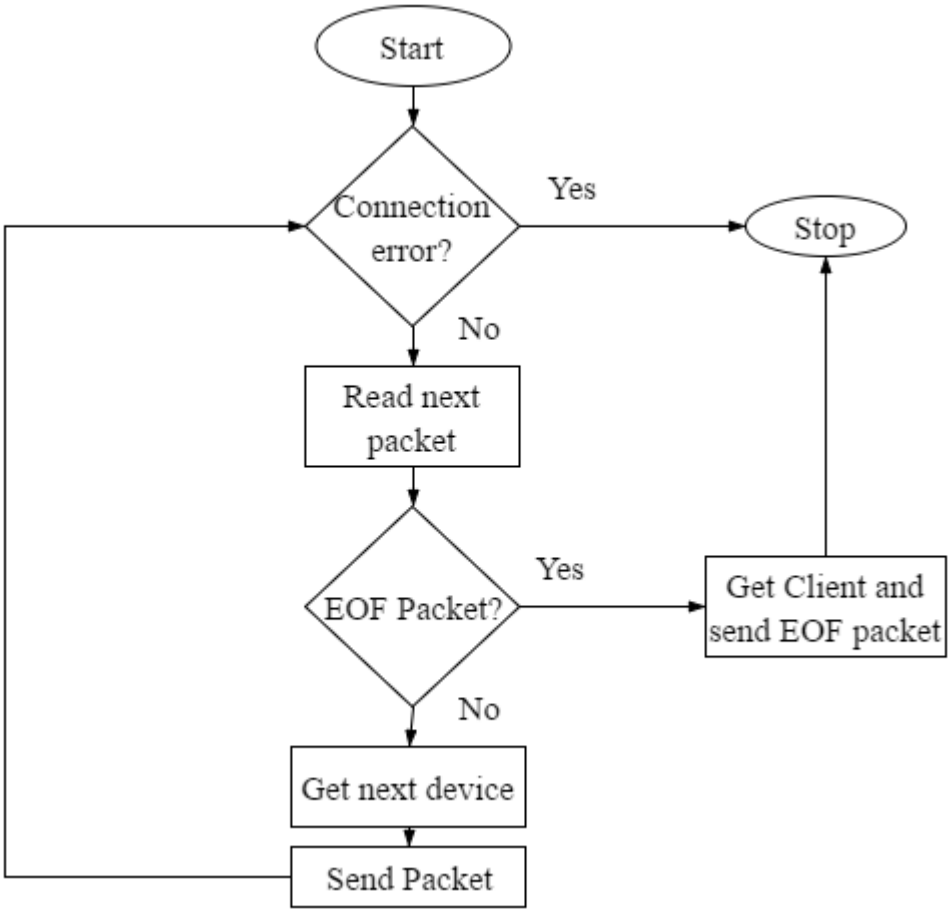

Figure 6.7: Flow chart of Frame Distributor

The Frame Distributor is implemented as follows

```
class frameDistributor implements Runnable {
    private Thread t;
    private String threadName;
    int tid;
    String videoLoc;
    transaction tn;
    reader r;
    ArrayList neighbours;
    ObjectOutputStream output;
    ObjectOutputStream out;
    frameDistributor(int id,String loc){
        this.threadName = "frame distributor thread";
        tid = id;
        videoLoc = loc;
        r = new reader(videoLoc);
        this.tn =
(transaction)Metadata.getInstance().transactions.get(tid);
        output = (ObjectOutputStream)tn.out;
    }
    @Override
    public void run() {
            int i = 0;
        for(;;) {
            if(tn.err)
```

```
                break;
            if(i++%100 == 0){
                Iterator iter = tn.devices.iterator();
                while(iter.hasNext()) {
                    Object device = iter.next();
                    ObjectOutputStream out = (ObjectOutputStream)device;
                    try {
                        out.reset();
                    } catch (IOException ex) {
                        tn.devices.remove(device);

Logger.getLogger(frameDistributor.class.getName()).log(Level.SEVERE, null,
ex);
                    }
                }
            }
            if(tn.devices.size() == 0)
                break;
            Iterator iter = tn.devices.iterator();
            while(iter.hasNext()) {
                Object device = iter.next();
                Object obj = r.readPacket();
                while(obj == null){
                    obj = r.readPacket();
                }
                if(obj != null) {
                    if(obj instanceof Integer && (int)obj == -1) {
                        obj = null;
                        configPacket cp = new
configPacket(configPacket.CMD.DONE);
                        try {
                            output.writeObject(cp);
                            cp = null;
                        } catch(IOException ex) {

Logger.getLogger(frameDistributor.class.getName()).log(Level.SEVERE, null,
ex);
                            break;
                        }
                        break;
                    }
                    else if(obj instanceof videoPacket || obj instanceof
audioPacket) {
                        try {
                            try {
                                ObjectOutputStream out =
(ObjectOutputStream)device;
                                out.writeObject(obj);
                            } catch(SocketException ex) {
                                tn.devices.remove(device);

Logger.getLogger(frameDistributor.class.getName()).log(Level.SEVERE, null,
ex);
                            }
                            obj = null;
                        } catch(Exception ex) {

Logger.getLogger(frameDistributor.class.getName()).log(Level.SEVERE, null,
ex);
                            break;
                        }
```

```
                    } else {
                        continue;
                    }
                }
            }
        }
        try {
            output.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        tn.err = true;
        System.out.println("Thread " +  threadName + " exiting.");
    }
    public void start() {
        System.out.println("Starting " +  threadName );
        if (t == null) {
            t = new Thread(this, threadName);
            t.start();
        }
    }
}
```

6.1.2 Client

The client has the Neighbor Manager, Plan Manager and the Video Player. The high level working of these components in unison is shown in the following flow chart, followed by the flowcharts and implementation of each of these individual components.

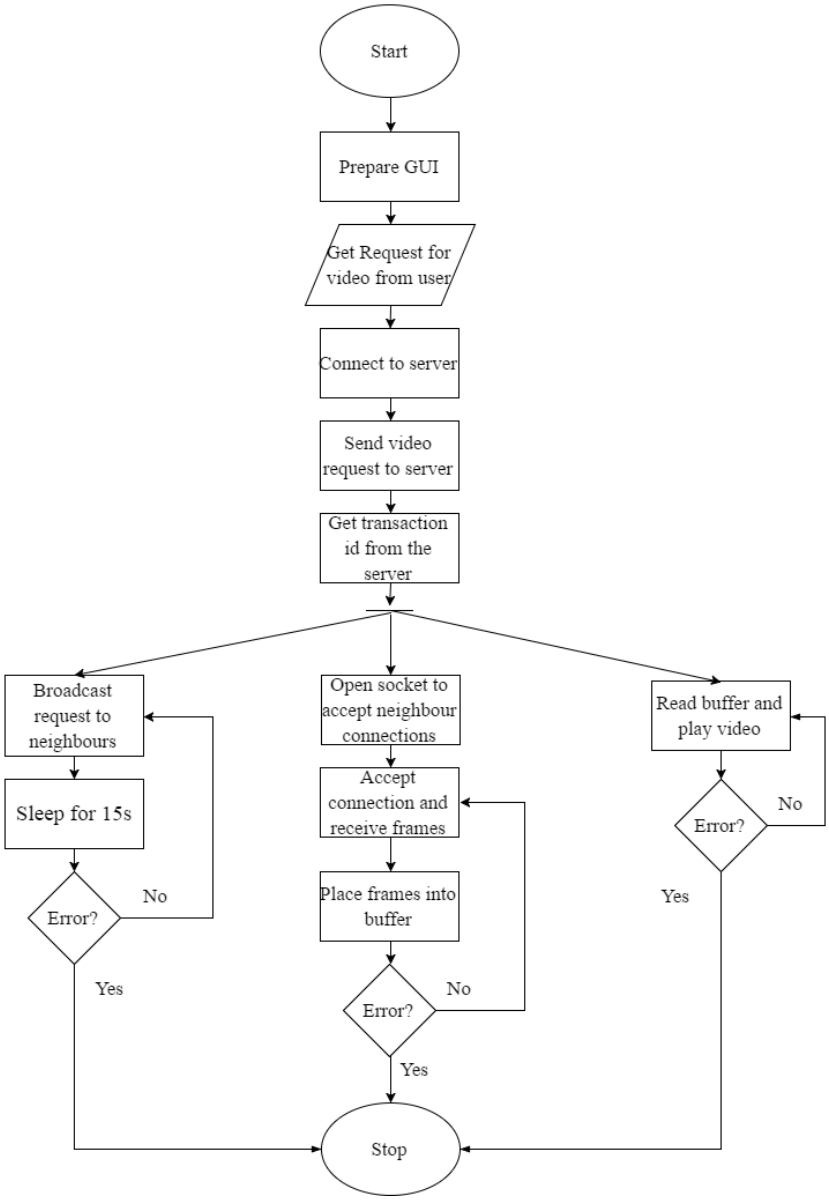

Figure 6.8: Flowchart of Client

a. Neighbor Manager

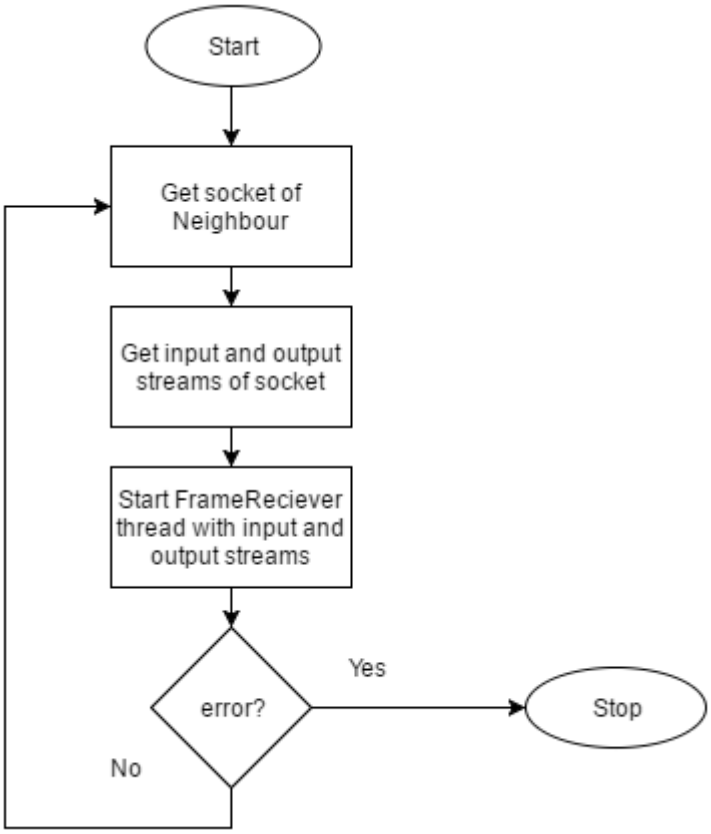

Figure 6.9: Fkowchart of Neighbor Manager

The Neighbor Manager is implemented as follows

```
public class NeighbourMgr implements Runnable {
    class neighbourHandler implements Runnable {
        private Thread t;
        private String threadName;
        Buffer buffer;
        final Client client;
        neighbourHandler(Buffer b, Client c){
            this.threadName = "Neighbour handler thread";
            this.buffer = b;
            this.client = c;
        }
        @Override
        public void run() {
            try {
                System.out.println("\nWaiting for neighbours on port " +
server.getLocalPort() + "...");
                for(;;) {
                    if(client.exitFlag)
                        break;
                    Socket clientSocket = null;
```

```
                    clientSocket = server.accept(); //initialize and close
this socket from neighbourMgr
                    ObjectInputStream in = new
ObjectInputStream(clientSocket.getInputStream());
                    ObjectOutputStream out = new
ObjectOutputStream(clientSocket.getOutputStream());
                    frameReceiver f = new frameReceiver(in,buffer,client);
                    f.start();
                }
            } catch(Exception ex) {

Logger.getLogger(neighbourHandler.class.getName()).log(Level.SEVERE, null,
ex);
            }
            System.out.println("Thread " +  threadName + " exiting.");
        }
        public void start() {
            System.out.println("Starting " +  threadName );
            if (t == null) {
                t = new Thread (this, threadName);
                t.start();
            }
        }
    }

}
```

b. Frame Receiver

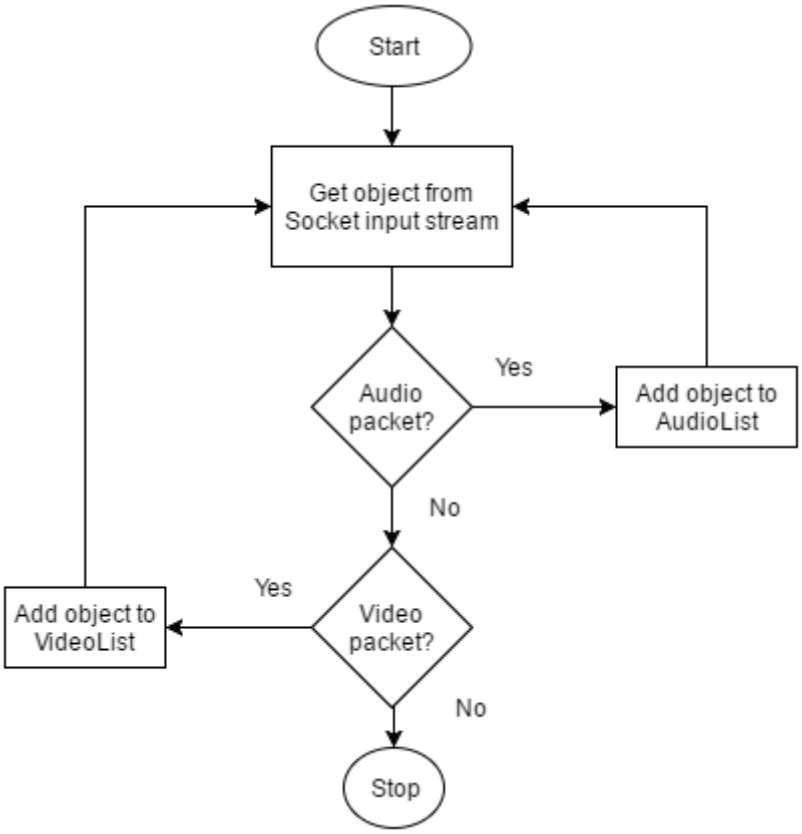

Figure 6.10: Flowchart of Frame Reciever

The Frame Reciever is implemented as follows

```
class frameReceiver implements Runnable {
        private Thread t;
        private String threadName;
        private ObjectInputStream input;
        Buffer buffer;
        final Client client;
        frameReceiver(ObjectInputStream in,Buffer b, Client c){
            this.input = in;
            this.buffer = b;
            this.threadName = "frame receiver thread";
            this.client = c;
        }
        @Override
        public void run() {
            for(;;) {
                if(client.exitFlag)
                    break;
                try {
                    Object obj = input.readObject();
                    if(obj instanceof videoPacket) {
                        videoPacket vp = (videoPacket)obj;
                        obj = null;
                        IVideoPicture videoPic = vp.getvideoPic();
                        synchronized (buffer.videoList) {
                            if(buffer.videoList != null) {
                                buffer.videoList.set(vp.id, videoPic);
                            }
                        }
                        vp.buf = null;
                        vp = null;
                    } else if(obj instanceof audioPacket) {
                        audioPacket ap = (audioPacket)obj;
                        obj = null;
                        IAudioSamples audioSample = ap.getAudioSample();
                        synchronized (buffer.audioList) {
                            if(buffer.audioList != null)
                                buffer.audioList.set(ap.id, audioSample);
                        }
                        ap.buf = null;
                        ap = null;
                    }
                } catch(IOException ex) {

Logger.getLogger(frameReceiver.class.getName()).log(Level.SEVERE, null,
ex);
                    break;
                } catch (ClassNotFoundException ex) {

Logger.getLogger(frameReceiver.class.getName()).log(Level.SEVERE, null,
ex);
                    break;
                }
            }
            try {
                input.close();
            } catch (IOException ex) {

Logger.getLogger(frameReceiver.class.getName()).log(Level.SEVERE, null,
ex);
```

```
            }
            System.out.println("Thread " +  threadName + " exiting.");
        }
        public void start() {
            System.out.println("Starting " +  threadName );
            if (t == null) {
                t = new Thread (this, threadName);
                t.start();
            }
        }
    }

    private Thread t;
    private String threadName;
    private int tid;
    Buffer buffer;
    final Client client;
    public static ServerSocket server;
    public NeighbourMgr(int tid,Buffer b,Client c){
        this.tid = tid;
        this.buffer = b;
        this.threadName = "request broadcaster thread";
        this.client = c;
    }

    @Override
    @SuppressWarnings("deprecation")
    public void run() {
        int i = 0;
        int DEFAULT_PORT = 2002;
        DatagramSocket socket = null;
        DatagramPacket packet;
        InetAddress group = null;
        try
        {

            server = new ServerSocket(2001,50);
            server.setSoTimeout(10000000);
            neighbourHandler nh = new neighbourHandler(buffer,client);
            nh.start();
            socket = new DatagramSocket();
            group = InetAddress.getByName("127.255.255.255");
            packet = new DatagramPacket(new byte[1], 1, group,
DEFAULT_PORT);
            configPacket cp = new configPacket(configPacket.CMD.NREQ);
            cp.setTransactionId(tid);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            for(;;) {
                if(client.exitFlag || client.doneFlag)
                    break;
                baos.reset();
                ObjectOutputStream oos = new ObjectOutputStream(baos);
                oos.reset();
                oos.writeObject(cp);
                oos.flush();
                byte[] buf = baos.toByteArray();
                packet = new DatagramPacket(new byte[1], 1, group,
DEFAULT_PORT);
                packet.setData(buf);
                packet.setLength(buf.length);
                socket.send(packet);
```

```
                Enumeration<NetworkInterface> interfaces =
NetworkInterface.getNetworkInterfaces();
                while (interfaces.hasMoreElements())
                {
                    NetworkInterface networkInterface =
interfaces.nextElement();
                    List<InterfaceAddress> interfaceAddresses =
networkInterface.getInterfaceAddresses();
                    Iterator<InterfaceAddress> it =
interfaceAddresses.iterator();
                    while(it.hasNext()) {
                        InterfaceAddress ia = it.next();
                        InetAddress broadcastAddr = ia.getBroadcast();
                        if(broadcastAddr != null) {
                            System.out.println("Broadcasting on
"+ia.getAddress().getHostAddress());
                            packet = new DatagramPacket(new byte[1], 1,
broadcastAddr, DEFAULT_PORT);
                            packet.setData(buf);
                            packet.setLength(buf.length);
                            socket.send(packet);
                        }
                    }
                }
                buf = null;
                Thread.sleep(3000);
            }
            socket.close();
            server.close();
        }
        catch( Exception ex )
        {

Logger.getLogger(NeighbourMgr.class.getName()).log(Level.SEVERE, null, ex);
            System.out.println("Problem creating socket on port: " +
DEFAULT_PORT );
        }
        System.out.println("Thread " +  threadName + " exiting.");
    }

    public void start() {
        System.out.println("Starting " +  threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start();
        }
    }
```

c. Neighbor

The high level working of the neighbor is depicted below.

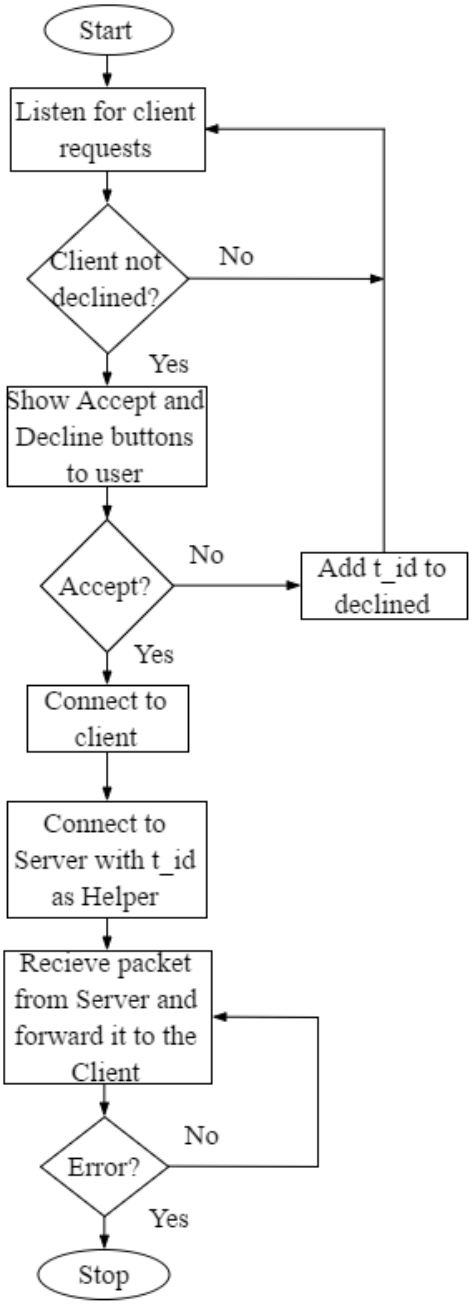

Figure 6.11

The Neighbor is implemented as follows

```
public class Neighbour {
    private static JFrame mainFrame;
    private static JPanel controlPanel;
    private static JButton acceptButton;
    private static JButton declineButton;
    private static JLabel state;
    private static JLabel request;
    private static boolean helping = false; //used in proxy class consider
making this volatile as well and extend base class
    private static ArrayList declined = new ArrayList();

    class Proxy implements Runnable {
        private Thread t;
        private String threadName;
        String address;
        int tid;

        Proxy(int id,String addr){
            this.address = addr;
            this.threadName = "Proxy handler thread";
            this.tid = id;
        }

        @Override
        public void run() {
            try {
                configPacket cp = new configPacket(configPacket.CMD.HLPR);
                cp.setTransactionId(tid);
                String path =
Neighbour.class.getProtectionDomain().getCodeSource().getLocation().getPath
();
                String decodedPath = URLDecoder.decode((new
File(path)).getParentFile().getPath() , "UTF-8");
                decodedPath += "/assets/server.txt"; //UNIX path
                //decodedPath += "\\assets\\server.txt"; //windows path
                //statusLabel.setText("Status bar:\n"+"looking up server ip
address from "+decodedPath);
                //System.out.println(decodedPath);
                FileReader fr = new FileReader(decodedPath);
                BufferedReader br = new BufferedReader(fr);
                String serverName = br.readLine();
                //String serverName = "192.168.0.108";
                String clientName = address;
                int sport = 2000, cport = 2001, i = 0;
                System.out.println("Connecting to " + serverName +" on port
" + sport);
                System.out.println("Connecting to " + clientName +" on port
" + cport);
                Socket server = new Socket(serverName, sport);
                Socket client = new Socket(clientName, cport);
                System.out.println("Just connected to " +
server.getRemoteSocketAddress());
                System.out.println("Just connected to " +
client.getRemoteSocketAddress());
                ObjectOutputStream out = new
ObjectOutputStream(server.getOutputStream());
                out.writeObject(cp);
                ObjectOutputStream out2 = new
ObjectOutputStream(client.getOutputStream());
                ObjectInputStream in = new
ObjectInputStream(server.getInputStream());
```

```
                for(;;) {
                    i++;
                    try {
                        if(i%20 == 0)
                            out2.reset();
                        Object obj = in.readObject();
                        if(obj != null){
                            out2.writeObject(obj);
                        }

                    } catch (ClassNotFoundException ex) {

Logger.getLogger(Proxy.class.getName()).log(Level.SEVERE, null, ex);
                        break;
                    }
                }
            } catch(IOException ex) {
                Logger.getLogger(Proxy.class.getName()).log(Level.SEVERE,
null, ex);
            }
            helping = false;
            System.out.println("Thread " +  threadName + " exiting.");
        }

        public void start() {
            System.out.println("Starting " +  threadName );
            if (t == null) {
                t = new Thread(this, threadName);
                t.start();
            }
        }
    }

    private class ClickListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JButton btn = (JButton)e.getSource();
            if (btn.getText() == "Accept") {
                int tid = (Integer)btn.getClientProperty("tid");
                String addr = (String)btn.getClientProperty("addr");
                helping = true;
                Proxy p = new Proxy(tid,addr);
                p.start();
            } else if (btn.getText() == "Decline") {
                int tid = (Integer)btn.getClientProperty("tid");
                declined.add(tid);
                controlPanel.setVisible(false);
            }
        }
    }
}
```

Chapter 7

SOFTWARE TESTING

7.1 Module Testing

7.1.1 Client Module

Testcase ID	1
Testcase Name	Client TC1
Testcase Description	This testcase tests the Client's ability to display the video list, establish connection to the server and play the video
Preconditions	• Server and Client are running • valid video files are present on the Server's filesystem
Input	User picks video from the list
Expected Output	• Display video list screen • Establish connection to server • Video starts playing

Testcase ID	2
Testcase Name	Client TC2
Testcase Description	This testcase tests the Client's ability to navigate back to the video list while playing to video
Preconditions	• Server and Client are running • Valid video files are present on the Server's filesystem
Input	User clicks back button while playing the video
Expected Output	• Display the video list screen • Stop playing the video • Send stop command to the server

Testcase ID	3
Testcase Name	Client TC3
Testcase Description	This testcase tests the Client's ability to stop playing the video and buffer when the next frame is not available.
Preconditions	• Server and Client are running • Valid video files are present on the Server's filesystem
Input	Next frame unavailable
Expected Output	• Pause the video and buffer • Start playing the video after the required frame is recieved

7.1.2 Neighbour Module

Testcase ID	4
Testcase Name	Neighbor TC1
Testcase Description	This testcase tests the Neighbor ‘s ability to determine if a request for transaction has not yet been declined and display a request
Preconditions	• Running neighbour, client and server systems. • Valid video files on the server
Input	Request received from client and it’s t_id not in declined list
Expected Output	• Display request dialog

Testcase ID	5
Testcase Name	Neighbor TC2
Testcase Description	This testcase tests the Neighbour‘s ability to add a request’s t_id if user decides to decline the request
Preconditions	• Running server, neighbour and client systems. • Decline list data structure on client system • Valid video files on server
Input	User clicks the decline button upon receiving the request
Expected Output	• t_id added to the decline list

Testcase ID	6
Testcase Name	Neighbor TC3
Testcase Description	This testcase tests the Neighbour‘s ability to determine if a request for transaction has been declined and ignore it.
Preconditions	• Running server, neighbour and client systems. • Decline list and transaction list data structure on client system • Valid video files on server
Input	Request received from client and it’s t_id in declined list
Expected Output	• Ignore request

Testcase ID	7
Testcase Name	Neighbor TC4
Testcase Description	This testcase tests the Neighbor ‘s ability to determine if a request has been accepted by the user
Preconditions	• Running server, neighbour and client systems. • Transaction list data structure on client system • Valid video files on server
Input	User clicks the accept button upon receiving the request
Expected Output	• Start proxy • Connect to server and client

7.1.3 Server Module

Testcase ID	8
Testcase Name	Server TC1
Testcase Description	This testcase tests the Server's ability to handle Client request
Preconditions	• Server and Client are running • Valid video files are present on the Server's filesystem
Input	Video request from Client
Expected Output	• Create new transaction • Initialize plan handler

Testcase ID	9
Testcase Name	Server TC2
Testcase Description	This testcase tests the Server's ability to accept Neighbor connections
Preconditions	• Server and neighbour are running • Valid video files are present on the Server's filesystem
Input	Helper request
Expected Output	Add Neighbor to transaction device list

7.2 Integration Testing

7.2.1 Client - Server

Testcase ID	1
Testcase Name	Client-Server TC1
Testcase Description	Client sends video request to server with no neighbours
Preconditions	• Server and Client are running • Valid video files are present on the Server's filesystem
Input	Video file request
Expected Output	Server returns meta data containing video size, audio size, number of audio and video frames and Transaction ID

7.2.1 Client-Neighbour

Testcase ID	2
Testcase Name	Client – Neighbour TC2
Testcase Description	Client broadcasts NREQ signal to neighbor which accepts request for bandwidth.
Preconditions	• Server, Neighbour and Client are running • Valid video files are present on the Server's filesystem
Input	Video request
Expected Output	Add Neighbor to transaction device list

Testcase ID	3
Testcase Name	Client – Neighbour TC3
Testcase Description	Client broadcasts NREQ signal to neighbor which declines request for bandwidth.
Preconditions	• Server and Client are running • Valid video files are present on the Server's filesystem
Input	Video request
Expected Output	Add neighbor to decline list

7.2.3 Neighbour-Server

Testcase ID	4
Testcase Name	Server – Neighbour TC4
Testcase Description	Neighbour receives the NREQ and accepts the request. The neighbour sends a signal as HLPR (helper) with T_ID to server.
Preconditions	• Server, Neighbour and Client are running • Valid video files are present on the Server's filesystem
Input	NREQ request
Expected Output	Server registers neighbour as a helper

7.3 System Testing

Testcase ID	1
Testcase Name	System testcase TC1
Testcase Description	Here we use one neighbour, which receives an NREQ request and is added to transaction list. The frames are divided between the neighbour and the client systems.
Preconditions	• Server, Neighbour and Client are running • Valid video files are present on the Server's filesystem
Input	Valid video file, Request for video
Expected Output	The number of rebuffer events are reduced relative to the test where no neighbours are used for streaming.

Testcase ID	2
Testcase Name	System testcase TC2
Testcase Description	This testcase measures the efficiency of the system with two or more neighbours, where the video frames are divided proportionately between all neighbours.
Preconditions	• Server and Client are running • A valid video file is present on the Server's filesystem
Input	Valid video file, Video request
Expected Output	The number of rebuffer events reduce as the number of neighbours increases upto a point of saturation.

Chapter 8

RESULTS

Screenshots of the execution follow.

The list of videos is displayed in the following manner on the client system. Once a video is selected the request for the video is sent to the server.

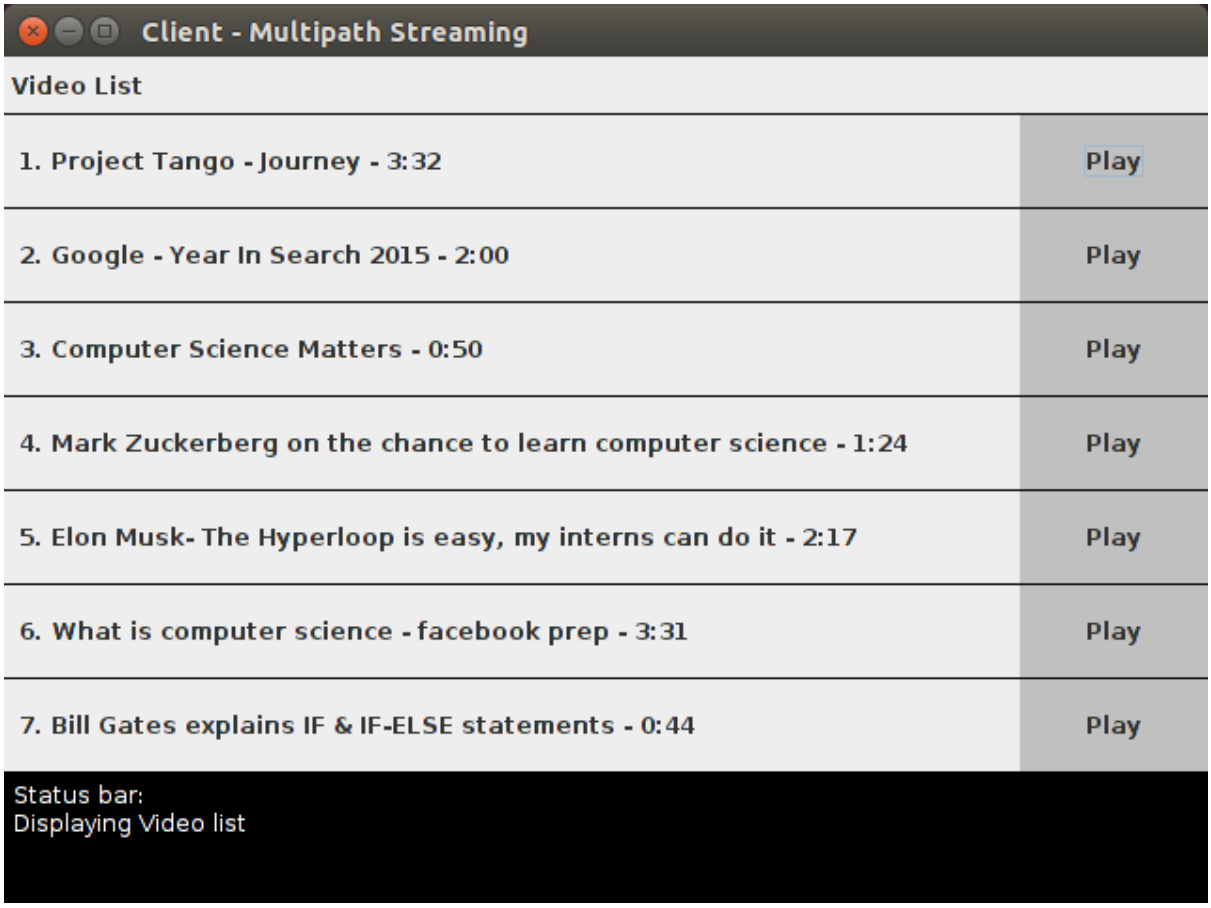

Figure 8.1: Client displaying video list

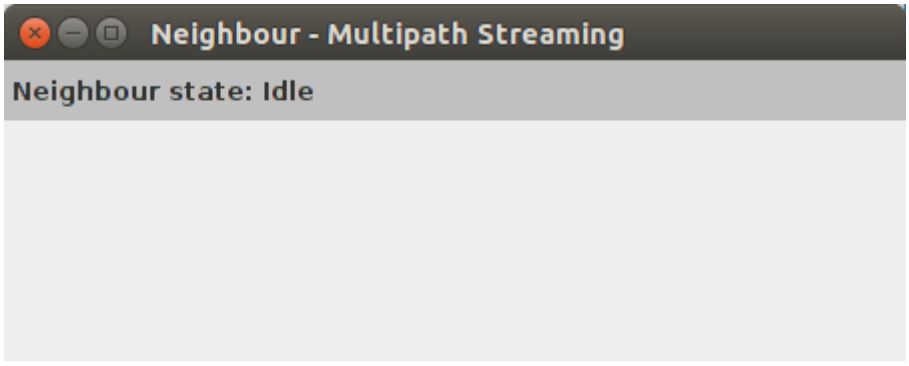

Figure 8.2: Initial Neighbour state

The request which is broadcast by the client is seen on all of neighboring systems as shown in the image. The neighbors can comply with the request or decline.

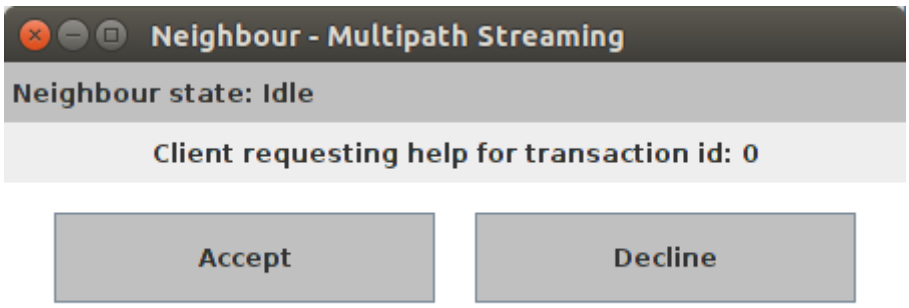

Figure 8.3: Neighbour receiving request

The image below shows the neighbour helping the client by streaming frames.

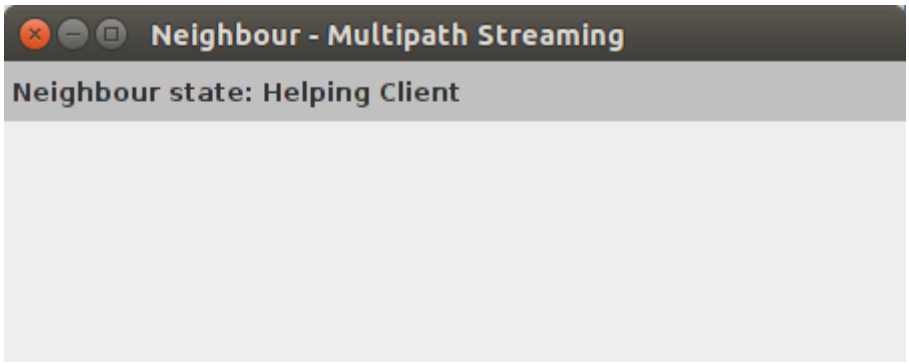

Figure 8.4: Neighbour accepting to help

The image below shows the client playing the video.

Figure 8.5: Client playing video

The image below shows the occurrence of a rebuffer event, the metric used to measure the performance of the system.

Figure 8.6: Client buffering video

The performance metric we use to measure improvement in video streaming experience is a **Rebuffer Event**. It is defined as follows.

"The number of rebuffer events is the number of times the Video Player stopped to rebuffer frames after it started playing."

A rebuffer Event is the most significant and noticeable trait of a video and is closely related to video viewing experience. In our experiments the independent variable will be the number of neighbors and the dependent variable will be the number of rebuffer events.

The following table summarises the results of the experiments.

Number of Neighbours	Buffer Time(m sec)	Number of Rebuffer Events
0	64922	20
1	56302	7
2	41110	0
3	31720	0

Table 8.1: Results

We observe a linear drop in the number of rebuffer events with increase in the number of cooperating neighbors.

Chapter 9

CONCLUSION

The popularity of video streaming systems has tremendously increased over the past few years. Despite its popularity, video streaming still remains a challenge in many scenarios. With limited broadband bandwidth at homes and 3G bandwidth in smart phones, the quality of video streaming suffers. To overcome this problem, we note two important network characteristics typical of most homes. First, there is a high density of Internet connections available in every neighborhood with many idle users. Although the available bandwidth at one Client is limited, the total unused bandwidth available in a neighborhood is high and can be aggregated together. Second, most of today‟s wireless devices have multiple interfaces that enable them to connect to nearby devices in an ad-hoc network at the same time they are connected to the Internet. CStream leverages the above two facts to aggregate Internet bandwidth for better video streaming.

The results show that when the Client and the Neighbors have equal bandwidth, the aggregate throughput achieved with CStream system increases linearly with the increase in the number of neighbors participating in the streaming. Playout time to stream and play the entire video also decreases multiplicatively as the number of neighbors increased. Similarly, the startup delay to play the first frame also decreased with the increase in number of contributing neighbors. A sharp decrease in the rebuffer events was observed as more Neighbors participated in video streaming. We ran experiments to verify the contribution of the Neighbors to video streaming and the results show that the ratio of the frames contributed by all nodes (both Client and Neighbor) is proportional to the available bandwidth. So bandwidth contributed by Neighbor is then limited by the minimum of the wired bandwidth and wireless throughput to the Client. CStream experimental results demonstrate how collaborative streaming improves the aggregate throughput and the quality of video streaming.

Chapter 10

FUTURE ENHANCEMENT

This chapter presents the possible extensions to CStream. Section 6.1 discusses a better frame distribution scheme that will improve the delivery of frames. Section 6.2 discusses plans for real world deployment and evaluation. Then, Section 6.3 and 6.4 details how CStream can be used with other video types and also streaming audio as a part of the video. Finally, Section 6.5 discusses security and incentives for CStream.

10.1 Better Frame Distribution

The current implementation of CStream adapts to the changes in the bandwidth, but when the bandwidth of the links is unequal, it leads to out of order delivery of frames. For example, if the Client is ten times faster than the Neighbor, and the first frame is assigned to the Neighbor, then according to current scheme it is likely that frames 2 to 11 will streamed via the Client link. Frame 1 will likely be received by the Client only after it has streamed frames 2 to 11. This out of order delivery of frames may impact the performance of video streaming and may result in increased rebuffer events. In such a case where the Client and Neighbor links have different bandwidth, then the frame distribution algorithm could assign the frames more intelligently.

10.2 Real World Deployment

A natural next step in evaluation is to measure the performance of CStream in a real-world setting. A real-world deployment, say in an apartment complex, will help study ISP diversity, their bandwidth optimizations and its effect on CStream performance. It will also help evaluate our assumption on the density of nodes and node idle time.

10.5 Incentives and Security

CStream assumes that all the Neighbors are willing to collaborate and are trusted, so there is no focus on incentives or security. One possible area of future work is to examine the security issues the system needs to address. This includes the Video Server encrypting and signing video frames and the Client verifying the content to prevent man-in-the-middle attacks. And maintaining a trusted set of neighbors and blacklisting malicious neighbors to control denial

of service attacks. Similarly it would be useful to come up with an incentive model to make CStream more practical and deployable. A simple solution for incentives is to implement a TFT (tit for tat) based scheme as in BitTorrents [LRLZ06]. In a TFT based scheme, a node gains credit when it uploads data to other peers and spends the credit to download data from other peers. A more complex solution is to design a micropayment based scheme similar to COMBINE [APRT07] to incentivize nodes to help. The payment scheme COMBINE uses includes a signed node of credit, termed an IOU (abbreviated from the phrase "I owe you") indicating the amount of payment made.

REFERENCES

[80211N] IEEE 802.11n-2009—Amendment 5: Enhancements for Higher Throughput. IEEE-Standards Association. 29 October 2009. doi:10.1109/IEEESTD.2009.5307322

[APRT07] G. Ananthanarayanan, V. Padmanabhan, L. Ravindranath and C. Thekkath. COMBINE: Leveraging the Power of Wireless Peers through Collaborative Downloading. In Proceedings of ACM Mobisys, San Juan, Puerto Rico, June 2007.

[AWW05] I. F. Akyildiz, X. Wang, and W. Wang. Wireless Mesh Networks: A Survey. In Elsevier Journal of Computer Networks, vol. 47, no. 4, pp. 445-487, 2005.

[CBB04] R. Chandra, V. Bahl and P. Bahl. Multinet: Connecting to Multiple IEEE 802.11 Networks using a Single Wireless Card, In Proceedings of IEEE Infocom, Hong Kong, March 2004.

[CR06] K. Chebrolu and R. Rao. Bandwidth Aggregation for Real Time Applications in Heterogeneous Wireless Networks, In IEEE Transactions on Mobile Computing, vol. 5, no. 4, pp. 388-403, April 2006.

[GALM07] P. Gill, M. Arlitt, Z. Li and A. Mahanti. YouTube Traffic Characterization: A View From the Edge, In Proceedings of ACM Internet Measurement Conference (IMC), San Diego, California, USA, October 2007.

[KLBK08] S. Kandula, K. Lin, T. Badirkhanli and D. Katabi. FatVAP: Aggregating AP Backhaul Bandwidth. In Proceedings of Networked Systems Design and Implementation (NSDI), San Francisco, CA, USA, April 2008.

[LRLZ06]	J. Liu, S. Rao, B. Li and H. Zhang. Opportunities and Challenges of Peer-to-Peer Internet Video Broadcast. In Proceedings of the IEEE, vol. 96, no. 1, pp. 11-24, January 2008.
[oAVI]	A Simple C# Wrapper for the AviFile Library. http://www.codeproject.com/KB/audio-video/avifilewrapper.aspx
[oEE09]	The Exabyte Era. http://www.cisco.com/web/IN/about/network/the_exabyte_era.html. Retrieved on 10 Sep. 2009.
[oORB]	Orb. http://www.orb.com.
[oWiki3G]	3G. http://en.wikipedia.org/wiki/3G.
[oYT]	YouTube. http://www.youtube.com.
[PMDC03]	R. Prasad, M. Murray, C. Dovrolis and K. Claffy. Bandwidth Estimation: Metrics, Measurement, Techniques and Tools. In IEEE Network, vol. 17, pp. 27-35, November 2003.
[RKB00]	P. Rodriguez, A. Kripa and E. W. Biersack. Parallel-access for Mirror Sites in the Internet. In Proceedings of IEEE Infocom, Tel Aviv, Israel, March 2000.