Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

OWDSTREAM: One-Way Delay Mechanism for Maintaining Latency Service Level Agreement

Author: Rishikumar Radhakrishnan (2701545)

1st supervisor:	Lin Wang
daily supervisor:	Vinod Nigage
2nd reader:	Henri E. Bal

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

February 23, 2022

"The cave you fear to enter holds the treasure you seek."
by Joseph Campbell

Abstract

Latency service-level agreement (SLA) is an agreement made by an application with a service provider to process a request within the given time period. The time period is usually the round trip time from the client's request to the server's response. In applications such as object tracking, facial identification where the heavy computational loads are offloaded onto a remote server a latency SLA is applied. These applications require the server to respond within the given time. Most often the servers speculate regarding the remaining time budget and process the request. This is due to the server not having the relevant information regarding the time taken for the request to reach the server from the client i.e., the One-Way Delay. In ideal cases the clocks on the client and server are synchronized and thus the One-Way Delay can be easily measured. If the One-Way Delay is available at the server, it is able to calculate the remaining time that is available for the application to maintain its latency SLA.

In this thesis, we propose Owdstream, a mechanism to measure the One-Way delay between the client and the server. We also design a framework such that it can be with used any client-server application to measure the One-Way Delay. Owdstream is able to calculate the clock offset between the client and the server by exchanging timestamp information through messages. These messages are piggy-backed on top of the request that is being sent to the server, therefore Owdstream does not create any new packets and thus does not contribute to the congestion on the network. Owdstream is based on the QUIC protocol. We perform various experiments under various conditions to test the validity of the framework in a synthetic test bed. Owdstream is able to consistently estimate the One-Way Delay within 3ms of the actual value and on average is able to achieve an estimation error rate of 5.43%.

Contents

List of Figures iii

List of Tables v

1 Introduction 1

2 Background 5

2.1 Clock Offset and Clock Drift 5

2.2 One-Way Delay: OWD 5

2.3 Precision Time Protocol: PTP 6

2.4 QUIC Protocol 8

3 Design & Implementation 11

3.1 One-Way Delay Mechanism 11

3.1.1 Calculating the Clock Offset 12

3.1.2 Calculating the One-Way Delay 13

3.2 Programming Interfaces to the Video Streaming Pipeline 14

3.2.1 Client-side Interfaces *quicconnectclient* 14

3.2.2 Server-side Interfaces *quicconnectserver* 16

4 Experimental Setup 19

4.1 Test Setup: Hardware and Software 19

4.2 Experiments 19

4.2.1 Measurement of Clock Offset 19

4.2.2 Measurement of One-Way Delay 20

4.3 Metrics 21

5 Evaluation 23
5.1 Measurement of Clock Offset 23
5.2 Measurement of One-Way Delay 27

6 Discussion 31
6.1 Limitations of the Evaluation Method 31
6.2 Limitations of the Design 31

7 Related Work 33
7.1 Applicability of Owdstream to Video Analytics 33
7.1.1 Computation on the Cloud 33
7.1.2 Computation on the Edge Devices 34
7.1.3 Hybrid Form of Computation 35
7.2 Time Delay 37
7.2.1 Round Trip Time 37
7.2.2 One-Way Delay 38
7.3 QUIC Protocol 38

8 Conclusion 41

References 43

List of Figures

2.1 One-Way Delay Components 6
2.2 Messages exchanged in PTP 8

3.1 Messages Exchanged to Calculate the Clock Offset 12
3.2 Video Stream Analytics based on Owdstream 15

5.1 Offset Measured: Variation of $Owdstream_{offset}$ and PTP_{offset} when $Synthetic_{offset}$ = 0 between the client and server. 24
5.2 Offset Measured: Variation of $Owdstream_{offset}$ and PTP_{offset} when $Synthetic_{offset}$ = 10ms between the client and server. The offset is induced at the server . . 25
5.3 Offset Measured: Variation of $Owdstream_{offset}$ and PTP_{offset} when $Synthetic_{offset}$ = 50ms between the client and server. The offset is induced at the server . . 25
5.4 Difference between Network Level and Application Level Time Stamping. . 26
5.5 Measured One-Way Delay between client and server with Bandwidth = 650Mbits/sec and datasize=50KB. 28
5.6 Measured One-Way Delay between client and server with Bandwidth = 250Mbits/sec and datasize=100KB. 29
5.7 Measured One-Way Delay between client and server with Bandwidth = 10Mbits/sec, datasize=100KB, and offset=10ms. 30

LIST OF FIGURES

List of Tables

5.1 Measured vs Actual Offset 23
5.2 Application Level Timestamping VS Network Level Timestamping 27
5.3 Measured One-Way Delay Values with offset=0ms 28
5.4 Measured One-Way Delay Values with varying offset values 29

LIST OF TABLES

1

Introduction

Latency service level agreements generally describes the services a network service guarantees within a specific time period [26]. These network services are generally applications that offload their computation onto a remote server. The time period is between the time at which a client sends a request to the server and the time at which a response is received at the client. The remote servers usually make their best effort to process a request and return the response within the given time budget. However, they are not able to guarantee the same since they do not have the relevant information regarding the time taken for the entire request to reach the server. This time taken is known as the One-Way Delay (OWD) [12], [3]. If the OWD is available at the server the application is able to determine the remaining time period and then decide on how to process the request efficiently. One of the applications of service level latency guarantees is in the field of video analytics.

Video analytics is the process of examining video input or video frames to perform certain functions. Video analytics is extensively used for identification in facial recognition systems, personal tracking systems, the state-of-the-art self-driving technology, object detection and in augmented reality [32], [35], [1]. Video as input generally produces a large amount of data. This video input is analyzed by using computer vision technology and the response is then used for taking the necessary actions [29]. Computer vision tasks are accomplished using Deep Neural Networks (DNNs). The DNN models are usually computationally heavy and require large amounts of resources [6]. Due to the data-intensive and resource hungry state of large-scale video processing, performing vision tasks are challenging. Most devices that produce video as data are low-cost, heavily constrained on resources and have processors that operate under low-power conditions. It is impractical to deploy the current DNN models on such low-power devices. An alternative is to deploy the DNNs on the cloud and offload these tasks onto the cloud server for analysis. The video input

produced are sent over the network to these cloud servers to be processed. As previously mentioned these applications have a requirement to maintain the latency SLA and due to the nature of these tasks, they also demand that the input be analyzed, and the result be sent back within a very small-time period or within a specified time budget usually in the range of milliseconds. This time budget starts when the video frame is sent as a request to the cloud server and stops when it receives the response at the client. [12]

The requests that are sent to the cloud server which are usually video frames are much larger in size when compared to the response that is sent back to the client. Thus, the request takes much longer to be transmitted completely when compared to the time taken by the response. The time budget can be split into three parts 1). time taken to send the request, 2). time taken to process the request by the application on the cloud server, and 3). time taken to send the response back to the client.

To process the requests within the specific time budget, the cloud server needs to determine the values of all the three parts of the time budget mentioned above. For the server to determine if a particular request is still relevant and can be processed within the given time, it must have the value of the time taken to receive that request and the time taken to process the same. The latter is deterministic and can be determined based on the application on the cloud server and also the size of the request [15], [22]. However, the time taken by the request over the network, or the OWD can vary a significantly as it depends on the bandwidth of the connection, congestion on the line or due to any other network issues. Thus, the OWD needs to be determined for each request separately due to the dynamic nature of the communication network.

We have formulated two research questions to find solutions to the challenges and limitations described for calculating the OWD. They are as following:

- How to estimate the clock offset between two systems?
- How to estimate the OWD with low overhead?

In this thesis, we propose Owdstream, a new framework which is built upon the existing protocol QUIC [31] that can measure the One-way delay (OWD) such that the application on the cloud server is able to determine the information regarding the remaining time budget available. This enables the application on the server to decide if the request is still relevant and needs to be processed or if it can be dropped and process the next request. Owdstream estimates the OWD and passes this information to the application for it to make its own decision. The OWD can be easily measured between two systems (client

and a server) if it is guaranteed that the two clocks are synchronized. If the clocks are synchronized, then the OWD is simply the difference between server timestamp (when the request was received) and client timestamp (when the request was sent). However, in most cases synchronization cannot be guaranteed and when two clocks are not synchronized, the desynchronization introduces a clock offset between them. If the OWD is calculated only using the client and server timestamps the resulting OWD would differ by the amount of the clock offset between the two systems [34].

Owdstream can calculate the offset between the client and server clocks and use it to determine the OWD between the client and server. The offset is measured on the client side and sent over to the server for each packet of the request. The clock offset is also piggy-backed on the existing transmissions and sent to the server. The generated timestamps at the client and server are used to calculate the Mean Path Delay (MPD). The MPD is the average time taken for the transmission of a packet from the client to the server and return back to the client as shown in Equation 2.1. The MPD is needed to find the offset between the two different clocks. The OWD is calculated using the offset, the sending time of the request on the client and the receiving time of the request on the server. Using the OWD the server can estimate the remaining time budget which the application on the server makes use of to determine the actions to be employed for that request.

The application on the server uses the OWD as feedback to decides the actions that needs to be taken on the request. The advantage of Owdstream is that there are no extra resources that are consumed as the mechanism to calculate the OWD is handled within the packets that are already being sent as part of the request to the server. This implies that there is *no congestion or delay* caused due to the usage of Owdstream over the network. Owdstream *does not demand any extra hardware* to be used to calculate the offset. In summary, the following contributions are made in this work:

- A simple yet powerful method to calculate clock offsets and consequently one-way delay
- Design and implementation of a framework known as Owdstream for streaming data based on QUIC for applications such as video analytics.
- Evaluation of our proof of concept of the framework on a synthetic test bed

2

Background

This chapter discusses the terms used throughout the thesis, their definitions and the protocols used in this thesis.

2.1 Clock Offset and Clock Drift

Clock Offset is the delay of a given clock when compared to a reference clock or a source clock. In this thesis the clock offset is the difference between the clocks on the client and server as measured on the client. If the measured clock offset is positive, it implies that the server's clock value is ahead of the client's clock value [34].

Clock drift is the event where the clock of a system does not run at the same rate as the clock of a reference system. Clock drift causes the two systems to eventually desynchronize over a time period. The is because the clocks on the two systems are running at different rates, thus the clock offset is calculated continuously and periodically to reduce this error. In this thesis we do not consider the clock drift since the clock offset is continuously being calculated the clock drift between the client and server is included in the clock offset. [34]

2.2 One-Way Delay: OWD

One-Way delay consists of three components 1) Application delay 2) Transmission delay 3) Propagation delay as shown in Figure 2.1 [16].

The first delay is caused by the application overhead it consists of the processing time, packet creation time, packet extraction time etc on the client and the server. This delay is dependent on the hardware or equipment the application is executed on and the efficiency of the application itself. The second delay is the time taken to push all the bits onto the

Figure 2.1: One-Way Delay Components

transmission link or to pull all the bits from the transmission link. This is dependent on the data size and the channel bandwidth. The third delay is the time taken for the packet to reach its destination from the host. This delay is dependent on medium on which it travels and the physical distance between the two nodes. OWD is the sum of all the three delays.

When a request is sent over the network the request is split into many packets depending on its size. The OWD of a request is the time di erence between the sending timestamp of the rst packet on the client and the receiving timestamp of the last packet on the server.

2.3 Precision Time Protocol: PTP

In this thesis we calculate the MPD and the o set using the same formula as that of PTP. In this section we discuss how PTP works and how PTP is incorporated within Owd-stream. The Precision Time Protocol (PTP) is a networking protocol that is used for clock synchronization between computer systems over the network. PTP can estimate and

synchronize the o set between two clocks within a sub-microsecond range. The protocol is implemented as a master-slave model where the slave receives messages containing timestamps from the master to calculate the o set. PTP can be used to calculate the o set and achieve synchronization without the use of external hardware such as boundary clocks or transparent clocks in case of a symmetric network i.e., the number of hops to the slave from the master is the same as the number of hops from the slave to the master. However, in cases of asymmetric connections external hardware such as transparent and boundary clocks must be used to achieve synchronization. [33]

The basic functionality of PTP is achieved by exchanging messages between the master and slave. The messages are sync, follow up, delay req, and delay response as shown in Figure 2.2.

The master clock initiates the exchange of messages. The master clock sends a sync message to the slave and stores the timestamp T_1 of when the sync message was transmitted. The timestamp T_1 is captured at the network level to ensure its accuracy. The master then sends a follow up message which contains the timestamp T_1 to the slave. The slave will record the timestamp T_2 at the network level when it receives the sync message from the master. The slave, once it receives the follow up message will send the delay request message to the master. The slave will also record the timestamp T_3 when the delay request message is transmitted to the master. Once the delay request message arrives at the master it records the timestamp as T_4 and transmits the timestamp T_4 to the slave in the delay response message. This process is shown in Figure 2.2

Once the delay response messages is received, the slave now has all the 4 timestamps that were generated. The slave is then able to calculate the mean path delay (MPD) using the formula shown in Equation 2.1. From the MPD we can calculate the o set using the formula shown in Equation 2.2. The Equations 2.1 and 2.2 are only applicable when the connection is symmetric i.e., the time taken for the packet to travel from the client to the server is equal to the time taken for the packet to travel from the server to the client.

$$MPD = ((T_2 - T_1) + (T_4 - T_3))/2 \tag{2.1}$$

$$Offset = T_2 - T_1 - MPD \tag{2.2}$$

Once the o set is calculated, the slave device will adjust its own system clock with the value of the o set in speci c time intervals and thereby synchronize itself with the master clock. The PTP application is not used directly because it would introduce new packets in

Figure 2.2: Messages exchanged in PTP

the transmission which would contribute to the congestion and the application would cause an overhead on the available resources in the client and server. Owdstream also utilizes the same theory of PTP to estimate the o set. By doing so we introduce the limitation that Owdstream can only be used in a symmetric network.

2.4 QUIC Protocol

Quick UDP Internet Connection protocol or QUIC is designed to improve latency and e ciency while also able to create multiple streams of data and provide with encryption. QUIC takes the best features of TCP connections and TLS encryption and builds it on top of UDP. One of the advantages of the QUIC protocol is that it is built entirely in the user space or application level which implies it is more exible and any changes can be deployed easily without much change in network devices or hardware

One of the reasons to use the QUIC protocol is its ability to create multiple streams for data transfer. We take advantage of this by sending the requests over multiple streams so that the consecutive requests are not blocked. QUIC only guarantees the same order of the requests as seen on the input when they are sent over a single stream and does not guarantee the same order when sent over multiple streams. Thus this needs to be handled in our own implementation.

We implement the QUIC protocol by using an open-source implementations of QUIC in Python known as aioquic [2]. aioquic is an event driven implementation of QUIC. aioquic will provide the user with the QUIC stack (i.e., the transport layer). Thus, it is up to the user to implement the application layer protocol on top of the quic stack. Due to this feature, we can implement our own application layer protocol to send the requests to the server while also allowing us to calculate the o set and the OWD.

There are many open-source implementation of the QUIC protocol but aioquic was chosen for the following reasons.

- aioquic is one of the most complete implementations, easy to use and is documented extensively by the developers providing working examples of a simple HTTP3 application, therefore it does not take much time to implement an application on aioquic.
- aioquic receives extended support from the developers, and it is constantly updated to re ect the changes in the QUIC draft
- As it is implemented on python it is lightweight and incredibly adaptive to suit the user's need.

Since aioquic is used in the implementation we do not need to worry about congestion control, packet loss, re-transmission etc. as it shipped with all these features. Our implementation is only the application level of this protocol and thus does not threaten the underlying implementation of QUIC. Thus there were no experiments conducted to test and evaluate the congestion control and re-transmission features of aioquic.

3

Design & Implementation

In this chapter we discuss the design and the implementation of the proposed Owdstream. The chapter is split into two where we first discuss the design of the one-way delay measurement mechanism and second we discuss how to incorporate Owdstream in an existing video streaming pipeline. The design and implementation is discussed in the context of video stream analytics to make it more convenient to the reader. In this case the request sent by the client is a video frame and the response is the analysis by the application on the server of the sent video frame. Thus we look at how in a video streaming pipeline the clock offset and OWD is measured.

3.1 One-Way Delay Mechanism

In this section we first identify the difference between a video frame and a packet. A video frame is generally in the range of 50KB-500KB depending on the frame resolution and frame encoding scheme. The video frame is the input in the application layer of the client. When a video frame is sent it split into many packets depending upon the size of the frame and maximum packet size allowed on the protocol. aioquic has the maximum size of a packet as 1200 bytes. Thus a 50 KB video frame will be sent across to the server in approximately 42 packets.

The clock offset is measured with respect to the packets whereas the OWD is measured with respect to the video frame. The OWD of the video frame when the clocks are synchronized is simply the time difference between the sending timestamp of the 1^{st} packet on the client side and the receiving timestamp of the 42^{nd} packet on the server. The OWD is calculated using the following steps.

Figure 3.1: Messages Exchanged to Calculate the Clock O set

- Measure and send the estimated clock o set and the timestamp of the rst packet in the video frame from client to server
- Measure the timestamps of the received packets on the server.
- Use the timestamp of the nal received packet in the video frame on the server, timestamp of the rst packet on the client, and the o set to calculate the OWD.

3.1.1 Calculating the Clock O set

The o set is calculated by exchanging timestamps between the client and server as show in Figure 3.1. As previously mentioned, when a video frame is sent to the server it is sent by splitting them into packets. We take advantage of this process and piggyback on these packets to exchange our timestamps and o set.

On the client side the sending timestampT_1 is captured and stored when a packet is being built in the aioquic implementation. Along with this timestamp, the previously measured o set is also sent across. However, at the initial stage the value of o set is zero

and is not considered by the server. The timestamp is piggy-backed on the existing data of the video frame and sent along with the packet.

On the server once the packet containing timestamp T_1 is received it generates a receiving timestamp T_2 to store for future use. The server parses through the data and gets the value of T_1.

Once the server has received the packet it sends an acknowledgment back to the client for the specific packet. Like the client the server will also generate a sending timestamp T_3 when building the acknowledgment packet to be sent. The timestamps generated are also sent back to the client along with acknowledgment.

When the acknowledgment reaches the client the final receiving timestamp T_4 is captured. The data is parsed and the timestamps T_2 and T_3 are retrieved. The client now has access to these four timestamps and uses it to calculate the MPD using the formula in Equation 2.1. Once the MPD is calculated the offset is determined using the formula in Equation 2.2.

Once these steps are completed, they are repeated until the end of transmission, and the new offset is sent to the server along with the new T_1 timestamp for the next packet. The server now has the offset value and stores it. This offset value represents the amount by which the clock in the server is ahead or behind the client clock.

The offset history is stored on the server. The server calculates the average offset value depending on the history window. The history window is a parameter that indicates the maximum number of offset values to be included while calculating the average offset. The size of the history window can be any positive integer value starting from 1. The size of the history window will influence the accuracy of the offset measurement. The default value of the history window is set as 15 to get the maximum accuracy in offset measurement.

The offset values sometime have a sudden peak due to the network conditions and other external factors. In this scenario the offset value varies quite a lot from the mean offset value. Using the point values of the offsets in the calculation would result in errors. The server will discard the offset values that are greater than some threshold value. In our case the threshold value was decided empirically and is the mean offset value $\pm$ 2.5ms.

3.1.2 Calculating the One-Way Delay

The server now can quantify the amount by which it is desynchronized from the client using the history of offset values. The OWD of a video frame is the time taken for all the packets in a video frame to reach the server.

The sending timestamp T_S is generated for a video frame when the data to be sent is received by the framework on the client side. This is because from the point of view of a user the overhead caused by Owdstream or the application delay also needs to be recorded. The T_S is piggy-backed on the video frame data even before it is split into packets and sent across the connection. As previously mentioned each request or video frame is sent in its own separate stream in aioquic. The client will set the endstream flag present in aioquic when the last packet of a video frame is sent. On the server side it knows that when a new stream is started the first packet it receives will contain T_S, this is stored in the server. The start of a new stream on the server also implies the start of a new video frame as well. To check if the entire video frame is received the server checks the endstream flag. If this flag is set it implies that the entire video frame is received. The video frame is then pushed to an internal video frame buffer and stored for the user to retrieve later. Owdstream will capture the timestamp of when the video frame was pushed onto the internal video frame buffer called T_R. Since the mean offset value calculated at the server represents the amount by which the server clock is ahead or behind with respect to the client clock depending on the sign of the offset. Thus if the server clock is ahead the offset will be positive and we can add the offset to the T_S or if the offset is negative we can subtract it from T_R. Thus we arrive at the equation given in 3.1. to calculate the One-Way Delay (OWD).

$$OWD = T_R - T_S + offset \tag{3.1}$$

3.2 Programming Interfaces to the Video Streaming Pipeline

In this section we discuss the design of the video streaming application on top of the QUIC stack implemented by aioquic. Figure 3.2 shows the outline of how the video streaming pipeline works. We also discuss the two new interfaces that are introduced.

Two new interfaces quicconnectclient and quicconnectserver are created for the user to establish a connection to a known server and transmit the data across. These interfaces will be discussed separately.

3.2.1 Client-side Interfaces quicconnectclient

To transmit the data to the server the interface quicconnectclient is called along with ip address and port number the application is executing on the server. aioquic has its own methods to establish a connection to the server these internal methods will not be discussed as they are available in the documentation for aioquic. quicconnectclient will internally

Figure 3.2: Video Stream Analytics based on Owdstream

call these methods to establish the connection. Once the connection is established two separate threads are started, the client sending thread (C_{ST}), and the client receiving thread (C_{RT}). The C_{ST} is designed to 1.) be non-blocking and asynchronous, 2.) help maintain data integrity, and 3.) maintain order of the video frames.

The C_{ST} will continuously probe an internal queue for the video frame data that needs to be transmitted. The application can populate this queue by using the interface send_frame(frame) which is available in the quicconnectclient This interface will get the data from the application and populate the sending queue along with the timestamp of when it received the data which is T_S. The send_frame(frame) which is a part of C_{ST} is an asynchronous and non-blocking interface and to implement this interface we maintain an internal bu er of outstanding frames.

The C_{ST} will get the data from this internal bu er, to maintain data integrity it will calculate the hash of the entire video frame. The hash of the video frame is obtained by using the sha256 hashing algorithm. The video frame is split into packets and data is added on top of these packets to support the clock o set calculations. The calculated hash is needed to ensure that the video frame is not modi ed on the server side.

The C_{ST} will also assign an ID known as Frame ID to each video frame which helps in re-ordering the frames on the server side if they are received out of order.

Once the hash value is calculated and it is added on top of the video frame data along with T_S and Frame ID and is sent to the aioquic network stack for transmission.

The C_{ST} will not wait for the transmission of the video frame to complete but returns and check if there are any more frames in the queue to be sent. This is done to ensure that we have a non-blocking send and to ensure that the sending queue does not fill up and run out of space. If there is any data in the queue the whole process will repeat. If there is no data in the queue C_{ST} will wait until there is data and will terminate only when the client closes the connection.

The C_{RT} works like the C_{ST}, this thread will listen for any data that the server sends back to the client. This is usually the response of the analysis done on the video frame. When the thread receives any data for the user it will populate a separate receiving queue. This queue can be polled by the application by calling the interface get_result(). When this interface is called it will returns the data in the queue to the application which usually contains the frame ID along with any response that the server adds.

The final interface that is available is the client_close() which is used to close the connection between the client and server, terminate the threads that were created, and clean up any resources that were allocated. When this interface is called and if there is no data in both the queues, then the connection is closed, the threads are exited, and clean up takes place. However, if there is any data in the queue that is yet to be transmitted or read by the application the client_close() interface will wait until they are empty and then proceed with closing the connection to the server.

3.2.2 Server-side Interfaces quicconnectserver

To start the QUIC server the interface quicconnectserver needs to be called with correct configuration parameter such as the port number. Since QUIC supports encryption, we can either run it in the encrypted mode or the unencrypted mode depending on the use case. When the server is started and after the connection is established with the client, two threads are started, the server receiving thread (S_{RT}) and the server sending thread (S_{ST}). The S_{RT} alone is started initially

The S_{RT} will wait for the video frames to arrive from the client. When the entire video frame is received the timestamps included for clock offset calculation are retrieved and stored separately. The T_S, Frame ID, hash value, offset, and the actual video frame data will now be available.

When the complete video frame is available the hash value is calculated using the sha256 hashing algorithm and matched against the hash value received from the client. This is

done to ensure the integrity of the data that is received. The video frame is then pushed onto the receiving queue in the order of the frame ID for the application to retrieve. When the frame is pushed to the queue, the timestamp T_R is generated and the OWD is calculated using the methodology discussed in section 3.1.2. The OWD is also pushed along with the video frame to the queue. This timestamp T_R is generated such that it includes the overhead caused by Owdstream on the server side.

The video frames can be retrieved from the receiving queue by the application by calling the interface receive() present in the quiconnectserver When this interface is called it returns the video frame along with the frame ID and OWD. This OWD or per frame latency on the server side is then used by the application to check if it is possible to maintain the latency service level agreement. If this is not possible it implies that the frame is no longer relevant, and the frame is dropped and the next frame is retrieved for processing.

If the OWD or per frame latency is still within the time period of the service level latency guarantee the frame is passed to the application which is usually a DNN model for analysis. However the decision of dropping the video frame or processing the video frame is entirely on the end application on the server. Owdstream will only pass the OWD value and the video frame to the end application.

The S_{ST} is used to send the result along with the frame ID from the application back to the client or to notify the client if the frame is dropped by the server. The S_{ST} is only invoked when the server has received at least one entire video frame. The interface server_reply(fraemid,result) is called to send the result along with the frame ID to the client. Adding the frame ID as one of its parameters ensures that this interface can be called only after at least receiving one complete video frame.

4

Experimental Setup

In this chapter we discus the experiments that were conducted to validate the e ectiveness of Owdstream and the conditions under which the experiments were conducted. The experiments are split into two parts as the framework measures the One-way delay in a two part process. 1.) The o set measurements under various conditions 2.) OWD estimation using the measured clock o set

4.1 Test Setup: Hardware and Software

All the experiments were conducted on two AWS EC2 instances each with 8GB of RAM running ubuntu 18.04. The two EC2 instances were geographically located in the same center. Using traceroute we were able to determine that the number of hops between the two EC2 instances is one in either direction implying that the connection is symmetric. The latency was measured across the two devices using ping which indicated the Round Trip Time to be around 0.71ms and the bandwidth was measured using iperf3. The maximum bandwidth between the two systems was 650Mbps. However, in a real world scenario the maximum bandwidth will not be always available to the user. Python version 3.9 was used to run the framework. The latest version of aioquic as of December 2021 was used to build Owdstream.

4.2 Experiments

4.2.1 Measurement of Clock O set

In the rst part of the experiments we measure the accuracy of the o set calculation in Owdstream. One EC2 instance is setup as the client and the other is setup as the server.

From the client EC2 instance data is continuously sent to the server. Since the size of the data or the rate of data transmission does not a ect the measurement of clock o set in Owdstream we set the data size to 100KB and the interval between two data frames to 30ms.

To verify the o set value recorded by Owdstream, we created our own application in C that works based on the principles of the Precision Time Protocol. This application is able to take advantage of the Network level time-stamping feature that is available. As previously stated since the connections between the 2 EC2 instances are symmetric, this PTP application can provide a very accurate value of the clock o set. This clock o set is considered as the ground truth and compared against the one recorded by Owdstream. In the rst stage of this experiment on the same EC2 instance which is setup as the client in Owdstream, the PTP application is started parallelly with Owdstream as the master. In the EC2 instance that is setup as the server in Owdstream, the PTP application is executed in the slave mode. The value of the clock o set is not a constant value and varies with time. To verify if Owdstream is able to capture these changes, Owdstream and the PTP application must both be started simultaneously and only those values that were captured within the same time period are compared. This will measure the the naturally occurring o set between the two systems. The Owdstream and the PTP application are both executed for a minimum duration of fteen minutes to capture all possible changes in the clock o set.

In the second stage of the experiment we manually introduce a synthetic clock o set between the two systems by either increasing or decreasing the system time on the EC2 instance that is setup as the server. By doing so we are able to control the value of the o set. We ensure that the systems are synchronized before we induce the o set by using the application known as chrony [11]. We set the following values of o set -10ms, 1ms, 10ms, 20ms, 30ms, 50ms and 1s. In real world network conditions we do not expect the o set to be more than 1s thus we do not go beyond it as well [8]. For every value of the o set we repeat the experiment mentioned above to capture the o sets recorded by the Owdstream and the PTP application.

4.2.2 Measurement of One-Way Delay

In the second part of the experiments we verify the accuracy of the measurement of One-Way Delay produced by Owdstream. In this experiment we assume that Owdstream is used in a video streaming application, therefore the parameters of the experiments such as datasize and bandwidth is varied accordingly. To capture the One-Way Delay we set up

the two EC2 instances similar to that of the previous experiment. From the EC2 instance that is set up as the client we create a random data frame and send it to the EC2 instance that is set up as the server at an interval of 30ms between two data frames. This is similar to a video source providing the input at 30 frames per second.

A standard de nition video frame of size 720x480 with an 8-bit color depth will approximately be 400KB and after compression it will be around 50KB. We varied the size of the random data from from 50KB to 200KB where 200KB corresponds to a video frame of size 1920x1080 with an 8-bit color depth after compression.

To simulate network congestion we vary the bandwidth of the connection from 10Mbps to the maximum available bandwidth of 650Mbps. We expect the One-Way Delay to increase as we lower the bandwidth. Additionally we set the following o set values -10ms, 10ms and 20ms by changing the system time at the server to verify if Owdstream still produces the correct One-Way Delay. In this experiment we send ten thousand data frames or video frames to the server from the client. We compare the One-Way Delay produced by Owdstream with the Theoretical One-Way Delay. The theoretical OWD is discussed in section 4.3.

4.3 Metrics

To quantify the accuracy of the measured o set and One-Way Delay values we introduce the following metrics that capture the errors in the measurement.

E_{offset} measures the di erence between the o set produced by Owdstream and the o set produced by the the PTP application as shown in equation 4.1. Ideally we expect the value of E_{offset} to be close to zero.

$$E_{offset} = Owdstream_{offset} \quad PTP_{offset} \tag{4.1}$$

As previously mentioned the One-Way Delay can be easily measured if we are able to guarantee that the systems are synchronized with each other. In our experimental setup since we can control both the clocks of the client and server, we are either able to guarantee synchronization between the client and server or we are able to quantify the amount by which the client and server are not synchronized. Thus, we are able to measure the $Theoretical_{OWD}$ using the equation show in 4.2 where T_R is the timestamp of when the video frame is received at the server, T_S is the timestamp when the video frame is transmitted at the client and $Synthetic_{offset}$ is zero when the systems are synchronized and

$Synthetic_{offset}$ is equal to the manually induced Offset when the systems are not synchronized. The $Theoretical_{OWD}$ is considered as the true One-Way Delay and is compared against the OWD produced by Owdstream. We expect the OWD produced by the Owdstream to be close to the value of the $Theoretical_{OWD}$.

$$Theoretical_{OWD} = T_R - T_S - Synthetic_{offset} \tag{4.2}$$

We establish a lower bound for the OWD by calculating the Raw_{OWD} using the equation shown in 4.3. The datasize is known and is controlled as part of the experiment and the bandwidth is measured using iperf3. However, the measured bandwidth is not an accurate value as the bandwidth does not remain constant throughout the transmission and can vary significantly. Thus this approach is not accurate enough to be used in a real world setting. The Raw_{OWD} is the sum of the propagation delay and part of the transmission delay shown in Figure 2.1. We expect the OWD produced by Owdstream to be higher than that of the Raw_{OWD}. The lower bound also helps in understanding the overhead caused by the application i.e. the application delay as shown in Figure 2.1.

$$Raw_{OWD} = datasize/bandwidth + RTT/2 \tag{4.3}$$

5

Evaluation

5.1 Measurement of Clock O set

The goal of the experiments is to nd the accuracy of the o set reported by Owdstream. Table 5.1 shows the values for the average o set obtained when measured using the Owdstream between the client and server and the actual o set reported by the PTP application. The average o set values are calculated by taking the average of all the o set values reported by Owdstream and PTP. We expected the E_{offset} to be zero for all cases but from Table 5.1 it can be observed that this not true. The positive value of E_{offset} implies that the measured o set from Owdstream is more than the actual o set reported by the PTP application.

$Synthetic_{Offset}$ (ms)	$Avg\ Owdstream_{Offset}$ (ms)	$Avg\ PTP_{offset}$ (ms)	E_{offset} (ms)
0	2.977	-0.0528	3.03
-10	-7.184	-10.221	3.037
1	3.801	0.814	2.987
10	12.728	9.902	2.825
20	22.687	19.893	2.793
30	32.691	29.725	2.964
50	52.713	49.845	2.868
1000	1002.257	999.732	2.525

Table 5.1: Measured vs Actual O set

Additionally we see that the E_{offset} does not increase or decrease but stays within the range of 2.5ms to 3ms. To further explore the variations in the o set values we plot a graph

of the o set reported by Owdstream and the o set reported by the PTP application. In Figures 5.1, 5.2, and 5.3 the y-axis represents the o set values and the x-axis represents o set index i.e. when x=25 it implies that it is the 25^{th} o set calculated. From the gures we can see that the $Owdstream_{offset}$ varies a lot at the beginning and then able to achieve steady state once $x \geq 15$. This is because the average o set is calculated only after receiving 15 values of o set as mentioned in section 3.1.1. Since there no such correction in the PTP application all the values of the o set are recorded which is why we see spikes in the gures for the PTP o set.It can also be noted that the E_{offset} value remains within the range of 2.5ms to 3ms in these gures as well.

Figure 5.1: O set Measured: Variation of $Owdstream_{offset}$ and PTP_{offset} when $Synthetic_{offset}$ = 0 between the client and server.

The non-zero values of E_{offset} from Table 5.1 can be attributed to the di erence in the process of how the Owdstream captures the timestamps when compared to the PTP application. The Owdstream works in the application layer and captures the timestamps in the application layer as well. However, the PTP application executes in the application layer but captures the timestamps from the network layer as shown in Figure 5.4.

The captured timestamps from the PTP application is the exact time that a packet is sent or received by the system. This implies that T_1 timestamp captured by Owdstream is smaller than the T_1 timestamp captured by the PTP application or T_1 of Owdstream < T_1 of PTP . Similarly the T_2 timestamp captured by the PTP application is smaller than

Figure 5.2: Offset Measured: Variation of $Owdstream_{offset}$ and PTP_{offset} when $Synthetic_{offset}$ = 10ms between the client and server. The offset is induced at the server

Figure 5.3: Offset Measured: Variation of $Owdstream_{offset}$ and PTP_{offset} when $Synthetic_{offset}$ = 50ms between the client and server. The offset is induced at the server

Figure 5.4: Di erence between Network Level and Application Level Time Stamping.

the T_2 timestamp captured by Owdstream or T_2 of Owdstream > T_2 of PTP even though they represent the receive time of the packet. Similarly T_3 of Owdstream < T_3 of PTP and T_4 of Owdstream > T_4 of PTP. Due to this di erence of where the timestamps are captured there is a delay caused due to the packet moving from top to bottom or bottom to top for sending and receiving respectively as shown in Figure 5.4.

If Owdstream also captured the timestamps from the Network layer and if we calculate the o set using the formula shown in Equation 2.2 with the new network timestamps, the value of o set will be lower than the original value. To validate this we use tcpdump to capture the exact timestamps of the packets in the network layer and compare it against the timestamps captured at an application level for the QUIC framework. This comparison is shown in Table 5.2. From Table 5.2 we can see that o set is closer to zero when we use the timestamps from the network level to calculate the value of the o set.

This shift in the o set can be considered as an error in the Owdstream o set measurement. This error depends on how fast the application is able to push the packets above or below in the network stack. This is dependent on the performance of the application itself and the system.

Application Level Timestamping on QUIC				
T_1	T_2	T_3	T_4	O set (ms)
1644161394.2860	1644161394.2914	1644161394.2914	1644161394.2921	2.360
1644161394.3361	1644161394.3413	1644161394.3413	1644161394.3423	2.105
Network Level Timestamping on QUIC				
T_1	T_2	T_3	T_4	O set (ms)
1644161394.2870	1644161394.2890	1644161394.2919	1644161394.2920	0.95
1644161394.3381	1644161394.3409	1644161394.3415	1644161394.3421	1.10

Table 5.2: Application Level Timestamping VS Network Level Timestamping

5.2 Measurement of One-Way Delay

We evaluate the accuracy of the $Owdstream_{OWD}$ by comparing it against the $Theoretical_{OWD}$ de ned in Equation 4.2. The average of the measured $Owdstream_{OWD}$ values are shown in Table 5.3 and Table 5.4. In Table 5.3 as expected we see that the One-Way Delay increases as the size of the data increases. Furthermore we expected the $Owdstream_{OWD}$ to increase as the bandwidth is decreased but this was not true. The OWD for a datasize of 100KB only increased when the bandwidth is 10Mbps. For other values of the bandwidth there was not much change to the $Owdstream_{OWD}$.

The purpose of the Raw_{OWD} is to establish a lower bound for the OWD. The OWD produced by Owdstream must never go below the Raw_{OWD}. In Table 5.3 we can see that the $Owdstream_{OWD}$ is greater than the Raw_{OWD} for all scenarios. To further explore the variations in the OWD we plotted a graph with the $Owdstream_{OWD}$ and $Theoretical_{OWD}$ as shown in Figures 5.5, 5.6 and 5.7. In the gures the y-axis represents the OWD and the x-axis represents the data ID or the frame ID. During the experiment we send around 10000 frames to the server. However, in the graph we show only for 200 data frames to make it more readable. In Figures 5.5 and 5.6 we expected the $Owdstream_{OWD}$ and $Theoretical_{OWD}$ to have the same value but this is not the case, this will be explored further later in this section. We can see that the OWD is not a constant value and is continuously varying this implies that we need to constantly measure the OWD.

In Figure 5.5 the OWD varied from 17ms to 42ms whereas in Figure 5.6 the OWD varied from 35ms to 92ms. The varying nature of the OWD can be attributed to varying

bandwidth, packet re-transmission, congestion, queuing delays etc.

DataSize (KB)	Bandwidth (Mbps)	Raw OWD (ms)	Owdstream OWD (ms)	Theoretical OWD (ms)
50	650	0.9688	25.0843	22.3024
100	650	1.5842	44.4401	42.102
150	650	2.1996	60.6315	58.5497
200	650	2.815	80.8637	78.8009
100	10	80.3535	89.899	87.7223
100	50	16.3535	44.7149	42.39
100	100	8.3535	44.6286	42.2817
100	250	3.5535	44.2446	41.8743

Table 5.3: Measured One-Way Delay Values with o set=0ms

Figure 5.5: Measured One-Way Delay between client and server with Bandwidth = 650Mbits/sec and datasize=50KB.

In Table 5.4 we measured the OWD while also changing the o set value between the client and the server. As expected the OWD for a speci c datasize and bandwidth has a similar value when compared to the OWD in the Table 5.3. We expect the OWD to not

Figure 5.6: Measured One-Way Delay between client and server with Bandwidth = 250Mbits/sec and datasize=100KB.

change or be very similar because changing the o set value does not a ect the time taken to transfer the data across. Similarly we plot the varying OWD for an o set of 10ms as shown in Figure 5.7.Figure 5.7 also follows the same trends as that of Figures 5.5 and 5.6.

DataSize (KB)	Bandwidth (Mbits/sec)	O set (ms)	Raw OWD (ms)	Owdstream OWD (ms)	Theoretical OWD (ms)
100	10	10	80.3535	85.7042	83.7986
100	50	20	16.3535	44.2852	42.1154
100	100	-10	8.3535	42.2794	40.7788

Table 5.4: Measured One-Way Delay Values with varying o set values

To measure the accuracy of the OWD from Owdstream we compare it against the $Theoretical_{OWD}$. We expect both the values to be equal. However from Tables 5.4 and 5.3 we can see that average di erence between the $Owdstream_{OWD}$ and $Theoretical_{OWD}$ is around 2.3ms. This di erence between the $Owdstream_{OWD}$ and $Theoretical_{OWD}$ is the same as the error we get when we measure the clock o set in section 5.1. The clock

Figure 5.7: Measured One-Way Delay between client and server with Bandwidth = 10Mbits/sec, datasize=100KB, and o set=10ms.

o set values are used in calculating the OWD as shown in equation 3.1. The variation in the measurement of the $Owdstream_{OWD}$ when compared to the theoretical OWD is due to the error in measurement of the clock o set which is as a consequence of capturing the timestamps for Owdstream on the application level.

$$ErrorRate = ((Owdstream_{OWD} - Theoretical_{OWD})/Theoretical_{OWD}) \quad 100 \quad (5.1)$$

To quantify the accuracy of Owdstream we take the average of the $Theoretical_{OWD}$, the average of the $Owdstream_{OWD}$ and calculate the error rate using Equation 5.1. Ideally we expect the error rate to be zero. We are able to achieve an error rate of 5.43%

6

Discussion

In this chapter we look at some of the limitations or weakness of the proposed Owdstream. We look at some of the limitations that were introduced due to the design of the Owdstream and the weakness in the method of evaluation of the Owdstream. We try to put forward some solutions to the weakness as well.

6.1 Limitations of the Evaluation Method

Owdstream was only tested on synthetic test bed between two EC2 instances connected symmetrically with each other. Thus the performance of the Owdstream might di er when testing on a real wide area network.

The performance of Owdstream was only measured between a single client and server. There were no experiments to measure its performance when multiple clients are connected to the server. Although aioquic stack is able to handle multiple clients, Owdstream is not designed to handle it. Thus any applications that have multiple clients are not able to take advantage of Owdstream. However the implementation can be modi ed to include a key-value pair to keep track of each client and their own timestamps enabling Owdstream to handle multiple clients.

6.2 Limitations of the Design

Owdstream is built on the QUIC stack that is provided by the existing python implementation of QUIC known as aioquic. However, QUIC as a protocol is still under construction, Hence any updates to the QUIC protocol will imply modi cations to aioquic to match those changes.Thus Owdstream must also be modi ed to incorporate these changes and

has to be tested again to con rm that the modi cations have not broken the functionality of the framework.

Owdstream is designed such that it expects the request that is sent to the server to be at least 3 the maximum packet size of the QUIC protocol. This is because we need a total of 3 messages exchanged between the client and server to calculate the o set. If the size of the request is smaller than the maximum packet size of the QUIC protocol, we will only be able to calculate the OWD accurately when we recieve the 3^{rd} request from the client.

Owdstream inherently assumes that the connection between the client and server is symmetric in nature i.e. that the forward path is the same as the reverse path. This is due to Owdstream using the formula for MPD given in Equation 2.1. The formula is derived under the assumption that the connection is symmetric. Thus if Owdstream is implemented on an asymmetric network connection depending on the ratio of the latency of the forward path to the latency of the reverse path the accuracy of the measured OWD will drop signi cantly and will not produce the expected results.

There are two possible solutions to overcome this weakness 1.) make use of specialized hardware such as transparent clocks and boundary clocks to transfer the timestamp information accurately to the server [33], and 2.) to setup an external system to which the client and server can synchronize to with help of a synchronization protocol such as chrony [11]. Thus we can measure the o set of the client and server relative to this external system and thereby measure the OWD. Both the solutions rely either on specialized hardware or external systems thus making it more expensive to implement.

Owdstream cannot deliver highly accurate values of o set as it generates the timestamps it requires at the application layer as shown in Figure 5.4. Depending on the level of accuracy needed two conceivable solutions are 1.) we are able to determine the value of E_{offset} which is the error between the measured o set and actual o set by running Owdstream on a client and server that are synchronized. Thus we can introduce a correction factor which is equal to the value of E_{offset} since the two systems are synchronized. However this correction factor depends on the performance of the system and sometimes can be more if other processes are running on the client or server simultaneously. The correction factor can only be introduced under speci c conditions and accuracy can only be slightly improved. 2.) to move the processes of generating the timestamps to the network layer. By doing so we will be able to capture exact timestamps needed and produce highly accurate results. However by doing so we will introduce two new messages, follow up & delay response exchanges between the client and the server per o set calculation.

7

Related Work

In this section we look at other works that are available which would bene t from using Owdstream. We look at speci cally the eld of Video analytics to understand how Owdstream could assist in maintaining latency SLA or if they are able to take advantage of the OWD available for other purposes.

We also look at other methods proposed to calculate the OWD, the usage of OWD and other applications that are built on top of the QUIC protocol.

7.1 Applicability of Owdstream to Video Analytics

Video analytics can be split into three sections depending on where the computation or the computer vision tasks take place. Based on this criteria this section is divided into three 1.) Computational Tasks on the cloud 2.) Computational task on the edge devices 3.) Hybrid form of computation where the load is shared between them.

7.1.1 Computation on the Cloud

Cloud Computing models generally have a high accuracy as these models have large computational resources readily available. However this model relies heavily on having a reliable network to transfer the input from the IOT devices to the server. Video les are usually large and require stable connections which is not possible in most real-time scenarios. Thus most research in this eld is to overcome these constraints.

Kuntai du et al [14] have proposed a novel method called DDS- DNN Driven Streaming that is capable of changing the video streaming protocol depending on the feedback it receives from the server. The clients receives its input from the video camera and regularly sends low quality video frames to the DNN server for inference. The server then analyses

this low quality video frame using a DNN model. If the accuracy is not above a speci c threshold value then the server sends a feedback to the client. The client re-encodes the relevant regions in each frame in a higher quality and sends it back to the server. The video compression and streaming protocols are decided by the server when the accuracy from the low quality video frame is below a speci c accuracy threshold. By having the DNN model decide the quality of the video frame they were able to increase their accuracy by 9% and reduce the bandwidth usage by 59%. However, it can be deduced that due to re-transmitting a video frame there will be an increase in the latency of the response as well. The DDS model can incorporate the OWD provided by Owdstream and decide if it needs to maintain a high accuracy or a very low latency value. This will also lead to further reduction in the bandwidth usage.

Haoyu Zhang et al [37] the authors of VideoStorm: a model scheduler which allows users to submit queries regarding vision processes. The scheduler VideoStorm runs on the server side and will then identify a pro le which has optimal resource-quality trade o . For example a license plate reader model will need a high resolution video to achieve high accuracy whereas a car counting model will have higher accuracy for low resolution videos as well. The scheduler considers the resource-quality trade o and also the lag tolerance to choose a Pareto optimal setting. Here the lag tolerance is the amount of time the processing can be delayed and lag de ned as the time di erence between the last arrival of a video frame and the last processing of a video frame. However, VideoStorm can incorporate the OWD to better understand the lag tolerance. However, since the lag tolerance are usually around 10s to 10 minutes, incorporating the OWD would not produce a substantial bene t.

Similar to VideoStorm [37] we also have AWStream [36] which is able to achieve low latency and high accuracy. AWStream actively learns the Pareto-optimal setting and how to invoke the di erent degrading functions in video streaming. It also measures and adapts to the network conditions for example if the network bandwidth is low the adapter increases the degrading functions to decrease the data rate. AWStream can also use OWD as one of its parameter to monitor the network conditions. One-Way Delay gives a better understanding of the network conditions than the bandwidth and RTT.

7.1.2 Computation on the Edge Devices

In this model the input video is not directly sent to the server or it isn't sent at all. Some models run ltering systems for aiding in vision tasks or light weight DNN models on the edge devices which have accuracy above a threshold value and also is able to run in the

limited resource environment. This helps in overcoming the limitation of needing a high bandwidth network.

CONVINCE:a new approach for camera connected in a dense deployment who have a spatial-temporal correlation [30]. CONVINCE takes a new approach where it takes all the cameras as a collective entity thereby enabling collaborative video analytics. It is able to reduce bandwidth requirements and computational costs by removing frames which have redundant information when cross-checked between all the camera. Due to this approximately only 25% of the original frames are transmitted. It also improves the vision algorithms due to the collaborative nature of the cameras and thus also able to achieve an accuracy of 91%.

A model similar to CONVINCE is proposed by Yuanqi Li et al [24] an On-camera filtering system called Reducto which is able to dynamically adapts the filtering decisions according to the accuracy and feature types. This system is employed directly on the camera thus limiting the computational overhead and also the network end-to-end overhead as well. Reducto is able to provide a bandwidth saving of 21-86% and 50-96% reduced computation while also maintaining a high accuracy value. It also able to perform better than in backend processing when compared to Chameleon [20] a hybrid form of computation that is discussed later.

EdgeEye [25] a new framework by Peng Liu et al. In this framework the live video is sent to an EdgeEye server which provides the video analysis function based on DNNs. It provides a high level abstraction such that the devices can easily offload the input to the server without needing much computational resources. Since there is no transmission of the video to a data center there are no privacy concerns and encryption is also not needed as the video is analysed closer to the source.

These models address the issue of privacy as the videos are inferred locally and not stored in a data center. However these models have high hardware constraints and cannot be used when very high accuracy is required. Owdstream addresses these concerns of low power devices and hardware constraints by offloading them onto a cloud server. However, models that are designed as edge only computation are the least suitable to adopt Owdstream.

7.1.3 Hybrid Form of Computation

These models as the name suggests are a hybrid of both the edge and cloud computing models where the workload is shared between the two servers such that are able to maintain a high accuracy and also have minimum bandwidth usage.

7. RELATED WORK

Glimpse [9] is a real-time object recognition on mobile devices. Since object recognition models require high computational resources, Glimpse runs on the server for higher accuracy but when the latency of the connection between the mobile device and the server is high, glimpse uses an active cache of video frames stored on the mobile device for object tracking only using hints from the server that arrive intermittently. Glimpse computes trigger frames (frames in which the answer from the server and the local device are likely to di er) and sends only these frames to the server to reduce the network bandwidth. Glimpse can also take advantage of the OWD provided by Owdstream, however it would need to measure the OWD from the server to the mobile devices so that the mobile devices are aware of the change in OWD.

Chameleon [20] is a computational model which periodically adapts video analytics pipelines for a trade o between resource and accuracy. For example using high frame rate for slow moving tra c is not necessary as low frame rate will have a high accuracy however high frame rate is required in cases of high speed tra c to maintain a high accuracy.

Rocket [4] is a perfect example of the hybrid approach. The hybrid approach for video analytics spans across the edge and the server. Rocket looks for changes in consecutive frames and only transmit the changes to the server. The lightweight DNN on the edge is invoked for inference. When the lightweight DNN is not able to produce a result that is above the threshold value the heavy DNN on the server is invoked.

Clown sh [28] and Rocket [4] are quite similar to each other. Clown sh also uses a lightweight DNN on the edge and a heavy DNN on the server similar to Rocket. Only few subset of the frames are sent to the server thereby reducing the consumption of the network bandwidth and also the result shown is the fusion of the local DNN and the heavy DNN in the server. Clown sh is able to achieve a accuracy value higher than that of an edge only analysis but it is lower when compared to a cloud only solution which is expected.

Clown sh, Rocket and Chameleon can all bene t from measuring the OWD when it sends a video frame to the server. By incorporating Owdstream the models will also be able to guarantee a low latency response. These models can use OWD as a substitute for bandwidth estimation.

DeepMon [17] unlike the previous models does not transmit the video frame. DeepMon extracts the features from the video set and then sends these features to the cloud resources for analysing. This greatly reduces the network bandwidth and also the risk of privacy as the video is not sent and only the features are being sent. DeepMon outperforms all edge only form of computation models. DeepMon is one of the hybrid form models that would least bene t from measuring the OWD using Owdstream.

Hybrid form of computation models are the ones best suited for Owdstream. The OWD can be made available at the client side as well. Doing so, the client is able to decide if it can transmit the video frame to the server and expect a response within the given time period. If the OWD read by the client is too high it can decide to process the video frame locally. Similarly since a local inference is already available in the client the server can verify if it will be able to send the response within the given time period. If the server is not able to do so it can drop the request and ask the client to use its own local inference.

7.2 Time Delay

In this section we look at the di erent categories of time delay that is used to determine the quality of the network connection. We also look at their applications and other methods to measure them as well. Time delay can be categorised into two parts - The rst metric is Round Trip Time (RTT). This is the total time taken from the source to the destination and then back to the source. RTT is generally used in most congestion control algorithms and is easy to measure. RTT assumes that the time taken on the forward path is the same as that of the reverse path as well. However, this is not the case in some scenarios.

The second metric is One-Way Delay (OWD). This is the time taken for a packet to reach the destination from the source. OWD is harder to measure when compared to RTT due to clock synchronization and clock o set. It is shown that using OWD instead of RTT for congestion control we can reduce the bottleneck bandwidth by 16% [13]

7.2.1 Round Trip Time

As previosuly mentioned on of the applications of RTT is in congestion control. BBR [7] is a congestion control algorithm that uses throughput and RTT for congestion control. The RTT and Bandwidth is continuously probed and the RTT is kept at the minimum possible value and the throughput is kept at the maximum allowed value. Unlike loss based congestion control who do not operate at minimum RTT. BBR provides much better latency and also less number of packet drops.

TIMELY [27] is another congestion control algorithm based on RTT for data centers. TIMELY depends on accurate measurement of RTT. Hence few assumptions are made such as 1. NIC is able to generate a timestamp 2. The NIC is also able to generate ACK without the OS involvement 3. Separate priority queues for ACK. RTT is continuously measured and the gradient of the RTT is used to either decrease or increase the number of packets transmitted.

The advantage of using RTT is that they are very easy to measure as there are no synchronization issues. RTT measurement do not require specialized hardware.

7.2.2 One-Way Delay

As mentioned OWD is harder to measure when compared to RTT. Since the measurement is taken across two devices their clocks need to be synchronized or the clock o set must be known [13].

A method to nd the OWD is to synchronize the di erent clocks as in [34]. The two methods to provide clock synchronization is 1. External server based 2. End-to-End measurement based.

In an external server based method, an external global host provides the time information to all the hosts in the network enabling easy calculation of the OWD. Network Time protocol(NTP) and Global positioning system(GPS) are based on this idea. However setting up an external server is often cumbersome and expensive and thus this method is not widely used. This method also requires setting up an external hardware which is used as a reference for the local clock.

The End-to-End measurements based method estimates the clock o set between the source and destination and removes them. PTP which was explained in Chapter 2 uses end-to-end measurements in symmetric connection to calculate the clock o set. If the connection is asymmetric it is not possible to calculate the o set without the use of external hardware. The framework designed also assumes the connection is symmetric.

An end-to-end based measurement system to measure the OWD is proposed by Jin-Hee Choi et al [10]. They were able to measure the One-Way delay with a very high accuracy. The OWD was measured on both symmetric network connections and asymmetric network connections. The solution what was provided was tested only in a simulated network using NS2 [18]. There are also details not available regarding how the experiment was conducted.When we tried to replicate their work in a real network scenario we were unable to get the results they had obtained.

7.3 QUIC Protocol

In this section we look at the QUIC protocol and why we built the framework on top of the QUIC protocl. QUIC or Quick UDP Internet Connection protocol is designed to improve latency and e ciency while also able to create multiple streams of data and provide with encryption. It is also being debated if the QUIC protocol is part of the application layer

or the transport layer [23] [19]. QUIC is currently still under development and thus ever changing. [31]

Some of the key advantages of QUIC are:

- Connection establishment latency
- Multiplexing without Head of Line Blocking
- Stream and connection flow control
- Authenticated and encrypted header and payload
- Flexible congestion control

When evaluating the performance between QUIC and TCP. The QUIC protocol performs better than TCP in connection establishment due to its handshake design and also provides bette QoE when video streaming for high resolution videos. However TCP perform better in the event of packet re-ordering as it considers these packets to be lost and thus end up transmitting packets at a slower rate. [21]

QUIC is beneficial to video streaming when compared to HTTP adaptive streaming(HAS) over TCP. QUIC is able to overcome the Head of Line Blocking issue which is present in HAS due to it using a single stream TCP by using multiple streams of data transfer. QUIC also has very low connection latency as the handhshake is completed in 0-RTT. QUIC uses connection identifier (CID) over IP and port numbers for identification which helps in switching networks easily when compared to TCP [5].

8

Conclusion

In this paper we designed a framework Owdstream that is able to measure the clock o set and the One-Way Delay between a client and the server. The OWD depends on the system the applications run on, the communication medium, size of the data transmitted, bandwidth, and the physical distance between the nodes. We were able to calculate the One-Way Delay by exchanging timestamp information piggy-backed over the data that is being exchanged between the client and the server. Thus we do not introduce any new messages that contribute towards congestion or delays on the network. Owdstream can calculate the clock o set value without the use of any external hardware.

Owdstream is able to take advantage of the multiple streams that are available for transmission by the QUIC protocol by sending the data frames in multiple streams. We performed various experiments with varying workloads and conditions to validate and measure the performance of Owdstream. In our experiments we were able to measure the OWD within 3ms of the actual value of OWD to achieve an error rate of 5.43%.

Owdstream can be used in any application where there is a need to measure the One-Way Delay with little to no modi cation to the libraries. Owdstream designed is able to assist applications by reporting the OWD to the server. Hence, the application on the server can make better decisions regarding their own processing, In this way Owdstream can be of service to these applications to complete within their given time period and thereby maintain their Latency SLA guarantee.

References

[1] Tariq Abdullah, Ashiq Anjum, M Fahim Tariq, Yusuf Baltaci, and Nikos Antonopoulos . Tra c monitoring using video analytics in clouds . In 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing , pages 39 48. IEEE, 2014. 1

[2] aioquic . A library for QUIC in Python. https://github.com/aiortc/aioquic , 2020. 9

[3] Guy Almes, Sunil Kalidindi, Matthew J. Zekauskas, and Al Morton . A One-Way Delay Metric for IP Performance Metrics (IPPM) . RFC 7679, January 2016. 1

[4] Ganesh Ananthanarayanan, Victor Bahl, Landon Cox, Alex Crown, Shadi Nogbahi, and Yuanchao Shu . Video Analytics - Killer App for Edge Computing . In Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '19, page 695 696, New York, NY, USA, 2019. Association for Computing Machinery. 36

[5] Sevket Arisu and Ali C Begen . Quickly starting media streams using QUIC . In Proceedings of the 23rd Packet Video Workshop, pages 1 6, 2018. 39

[6] Ilai Bavati . Video Analytics at Scale: Challenges and Best Practices , 2021. 1

[7] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson . BBR: congestion-based congestion control . Communications of the ACM, 60(2):58 66, 2017. 37

[8] Qasim M. Chaudhari, Erchin Serpedin, and Khalid Qaraqe . On Maximum Likelihood Estimation of Clock O set and Skew in Networks With

Exponential Delays . IEEE Transactions on Signal Processing, 56(4):1685 1697, 2008. 20

[9] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan . Glimpse: Continuous, real-time object recognition on mobile devices . In Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, pages 155 168, 2015. 36

[10] Jin-Hee Choi and Chuck Yoo . One-way delay estimation and its application . Computer Communications, 28(7):819 828, 2005. 38

[11] chrony . Chrony: A versatile implementation of the Network Time Protocol (NTP) . https://github.com/mlichvar/chrony , 2017. 20, 32

[12] Luca De Vito, Sergio Rapuano, and Laura Tomaciello . One-way delay measurement: State of the art . IEEE Transactions on Instrumentation and Measurement, 57(12):2742 2750, 2008. 1, 2

[13] Carlo Demichelis and Philip Chimento . Rfc3393: Ip packet delay variation metric for ip performance metrics (ippm) , 2002. 37, 38

[14] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang . Server-Driven Video Streaming for Deep Learning Inference . In Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, pages 557 570, 2020. 33

[15] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace . Serving DNNs like Clockwork: Performance Predictability from the Bottom Up . In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 443 462. USENIX Association, November 2020. 2

[16] Ana Hernandez and Eduardo Magana . One-way Delay Measurement and Characterization . In International Conference on Networking and Services (ICNS '07), pages 114 114, 2007. 5

[17] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95, 2017. 36

[18] Teerawat Issariyakul and Ekram Hossain. Introduction to network simulator 2 (NS2). In *Introduction to network simulator NS2*, pages 1–18. Springer, 2009. 38

[19] Janardhan Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport; draft-ietf-quic-transport-24. *Internet Engineering Task Force: Newark, DE, USA*, 2019. 39

[20] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 253–266, 2018. 35, 36

[21] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols. In *Proceedings of the 2017 Internet Measurement Conference*, pages 290–303, 2017. 39

[22] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019. 2

[23] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery. 39

[24] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-Camera Filtering for Resource-E□cient Real-Time Video Analytics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 359–376, 2020. 35

[25] Peng Liu, Bozhao Qi, and Suman Banerjee. Edgeeye: An edge service framework for real-time intelligent video analytics. In *Proceedings of the 1st international workshop on edge systems, analytics and networking*, pages 1–6, 2018. 35

[26] J. Martin and A. Nilsson. On service level agreements for IP networks. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, 2, pages 855–863 vol.2, 2002. 1

[27] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015. 37

[28] Vinod Nigade, Lin Wang, and Henri Bal. Clownfish: Edge and Cloud Symbiosis for Video Stream Analytics. In *ACM/IEEE Symposium on Edge Computing (SEC)*, 2020. 36

[29] Iyiola E Olatunji and Chun-Hung Cheng. Video analytics for visual surveillance and applications: An overview and survey. *Machine Learning Paradigms*, pages 475–515, 2019. 1

[30] Hannaneh Barahouei Pasandi and Tamer Nadeem. Convince: Collaborative cross-camera video analytics at the edge. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 1–5. IEEE, 2020. 35

[31] Colin Perkins and Jörg Ott. Real-Time Audio-Visual Media Transport over QUIC. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, EPIQ'18, page 36–42, New York, NY, USA, 2018. Association for Computing Machinery. 2, 39

[32] Carlo S. Regazzoni, Andrea Cavallaro, Ying Wu, Janusz Konrad, and Arun Hampapur. Video Analytics for Surveillance: Theory and Practice [From the Guest Editors]. *IEEE Signal Processing Magazine*, 27(5):16–17, 2010. 1

[33] Vinay Shankarkumar, Laurent Montini, Tim Frost, and Greg Dowd. Precision Time Protocol Version 2 (PTPv2) Management Information Base. RFC 8173, June 2017. 7, 32

[34] Minsu Shin, Mankyu Park, Deockgil Oh, Byungchul Kim, and Jaeyong Lee. Clock Synchronization for One-Way Delay Measurement: A Survey. In Tai-hoon Kim, Hojjat Adeli, Rosslin John Robles, and Maricel Balitanas, editors, *Advanced Communication and Networking*, pages 1–10, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. 3, 5, 38

[35] Li Wang and Dennis Sng. Deep learning algorithms with applications to video analytics for a smart city: A survey. *arXiv preprint arXiv:1512.03131*, 2015. 1

[36] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 236–252, 2018. 34

[37] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 377–392, 2017. 34