WeSeeYou: Adapting Video Streaming for Surveillance Applications

Joshua Lloret, Robyn McCue, and Jie Wu
Department of Computer and Information Sciences
Temple University, Philadelphia, USA
{joshua.lloret, robyn.mccue, jiewu}@temple.edu

***Abstract*—Police departments and other law enforcement agencies have integrated a greater number of video cameras into their daily routines. This has introduced with it the problem of moving and processing vast amounts of video data. In this work, we take a look at some of the associated problems and our own attempts to address them. We first analyze the physical network and infrastructure at Temple University's main campus to determine the limitations we will meet in the real world. We investigate and implement a simple technique to transfer data across multiple wireless networks. Finally we look at different techniques of limiting the video we will transfer in the network switches to ensure that a video we want to prioritize reaches its intended destination in real time.**

Keywords*—*LTE, OpenFlow, REU, SDN, Wi-Fi, WiMAX

I. Introduction

Recently, increasing demand for oversight and accountability of law enforcement officers has resulted in police departments incorporating more video cameras into their arsenals of equipment.

Previous work has been done to take advantage of cameras on police vehicles and video processing to query license plate numbers for flags or related criminal activity[2]. The computing power that can be placed in a car is sufficient to recognize the numbers and letters from a license plate and subsequently run a query on a small local database, but it is not nearly powerful enough for more advanced video analysis techniques. Our group was interested in running the captured video through algorithms that process things like facial recognition and other computer vision tasks. Similarly, tasks that require access to large databases of information cannot be done locally on the car, and thus must be done on a remote processor.

Therefore, the video must be transmitted from the camera to a remote location. In order to determine how to best achieve this, we tested available wireless networks, including Wi-Fi and LTE. We investigated how to transfer the video streams using a simple transfer program written in Python. Finally, using the OpenFlow standard, we implemented a simple switch controller to manage the flow of video streams between the WAN and our server.

II. Testing

Transferring data across campus Wi-Fi is something that many students will tell you to be nearly impossible. Our approach was to use campus Wi-Fi when we could connect to it and other wireless spectrums like WiMAX or LTE when we could not. This would allow us to broadcast video streams from the vehicles continuously regardless of the performance of any specific wireless network.

	Avg Mbits/s	Mbits/s	Range
Wi-Fi (11n)	28.12	600	820ft (250m)
LTE	9.12	300	20mi (32km)
WiMAX	1.56	219	30mi (48km)

TABLE I: Comparison of the measured and theoretical download speeds of the different wireless technologies, as well as a listing of their theoretical range

In order to find out where we could connect and what kind of speeds we could expect from these connections, we ran a series of tests. The tests involved running an iperf server on a remote host and a client on a laptop computer. The laptop was then taken to different locations on campus and set to transfer either for 15 minutes continuously or for a fixed chunk of 300MB. We performed the 15 minute tests at stationary locations as well as while walking a route around campus. These tests allowed us to see the variability of throughput across a length of time and the simple numbers indicating how long it took to transfer x bytes.

Since we were interested in measuring throughput speeds when connected, we did not want the time spent negotiating a Wi-Fi connection to be a factor in our results. Therefore, we measured the Wi-Fi slightly differently from the LTE for our walking test. The reason for this is that the LTE connection was never dropped, even if throughput occasionally went to 0. However, the Wi-Fi connection was lost repeatedly, and this caused problems with physically reconnecting with the laptop as well as connecting the iperf client back to the server.

When trying to collect the Wi-Fi data for the walking test, we would often walk out of the range of the Wi-Fi router while the computer was still trying to negotiate its connection. Similarly, the iperf client was not able to recover from a lost connection to the server and would have to be restarted. To deal with these problems during testing, we decided to simply stop the clock and stop walking when we came within range of Wi-Fi and wait for it to connect. When the Wi-Fi was finished connecting and the iperf client had reconnected to the server, we restarted the timer and continued walking. This gives us a data set that tries to ignore the cost of Wi-Fi network switching and looks only at throughput of packets.

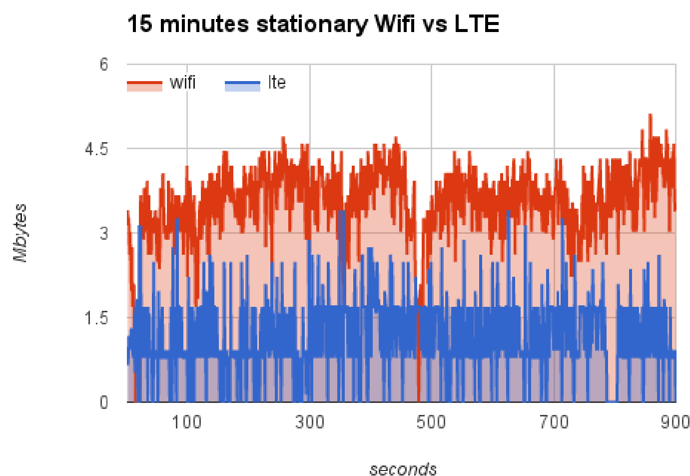

Fig. 1: A graph showing variability of packet throughput over 15 minutes of testing at a single location.

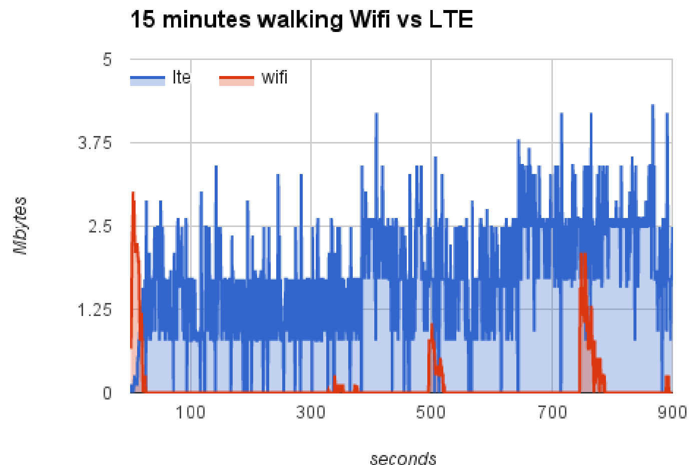

Fig. 2: A graph showing variability of packet throughput over 15 minutes of testing while walking along a route through campus.

Unlike iperf, our client continues streaming data without error after a connection is dropped and then reconnected. This is the subject of the next section.

III. Transfer

A simple transfer program was written in Python to transfer data across our variable networking setup. The host side of the program takes an output directory and an optional port number as input. The host then listens on that port for connections from a client.

The client takes a folder and host name and port number as input. First, the client tries to initiate a connection with the host specified at the given port number. After a connection is established, the client selects the first file from the folder and reads in the file's metadata. The metadata includes things like

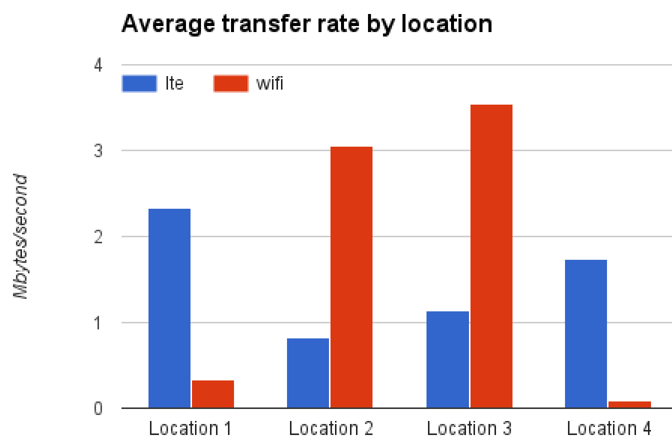

Fig. 3: A graph showing the average transfer rate at four different locations around campus.

filename and size. Next the metadata is sent as the first "chunk" of information from the client. When the server receives this information, it sends a message to the client indicating that the chunk was received. If the chunk is file metadata, the host creates a file with the given filename in the output directory.

Once the client receives a confirmation that the metadata was received, it opens the file, reads a chunk of a specified size, and sends that chunk over the network to the host. The host receives this chunk and sends the appropriate message back to the client. If the client does not receive an appropriate message from the host, either because the message was lost or because the original chunk was never received by the host, the client will wait 500ms and try to send the chunk again.

It is by this method that we ensure the entire file is copied from the client to the host and that none of the chunks are received out of order. This method also allows us to be blind to *how* the client is connected to the host: Wi-Fi, WiMAX, LTE, or even written message delivered via rickshaw, the means of connection does not matter.

This client host code can transfer static data, and it relies on transferring the file's size in the beginning. We are interested in transferring live video streams which will not have size information available. Luckily, this size information was only transferred for convenience, and adapting the code to transfer chunks of live stream data is possible.

More research and testing is planned to find the optimal chunk size. We will have to take into account the throughput of the network we are currently on and which data we hope to transfer when determining chunk size. Initially, chunk size will be something decided before we start the program and which will remain unchanged through the transfer, but a more complicated algorithm can be imagined which adapts the chunk size to the network's throughput and reliability.

For instance, a more reliable network could take larger chunk sizes; we can be more sure that our chunk will arrive and that we will receive a confirmation of its arrival from the

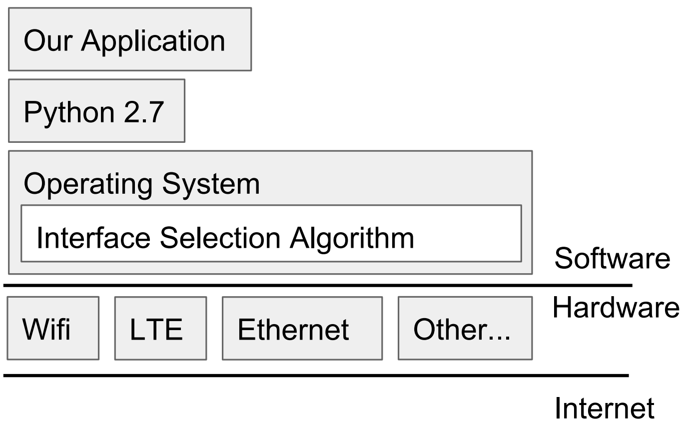

Fig. 4: A diagram of the technology stack on which our simple transfer code is written.

host. On the other hand, a less reliable network will necessitate smaller chunks; if we lose one it will be a minimal loss, and data can be sent in an instant. Considering throughput, if we have a slow network connection, we may reach a point of diminishing returns when shrinking chunk size; every time we send a chunk, there is the additional overhead of the host sending back a confirmation message.

While this program was originally fine for testing file transfers independent of network or platform, there are many other, more complicated algorithms we hope to expand this work with. There has been useful work done on transferring video using inter-layer network coding[3]. The algorithm described by Ostovari involves separating out and attaching more priority to the more important frames of the video. It has the added benefit that it is not blocking like our script, so it is free of the overhead of waiting for a response from the server before sending more video data.

As well as taking advantage of the frame data of certain encoding formats, we can take advantage of the data needed by our video processing host. If the host can only process a single frame of video every second, that means we only have to send, at max, a single frame a second from the capture point. Since captured video is normally around 30 frames per second, this could potentially cut video transfer down to a fraction of the cost.

Similarly, certain video processing techniques such as edge detection and character recognition are not as dependent on color information, so that can be cut down or stripped entirely from the video before transfer cuts down on the data again.

Lastly, there could be different “modes” or states of processing, where all captured video is transferred to the processing host at a lower quality, and when one video is determined to be more important, the host can send a message to that client to “upgrade” its video quality to allow for more detailed processing. All the other clients would “downgrade” their qualities accordingly.

IV. Switching

On the topic of data transfer, we would be remiss to leave out solutions in the switches between the host and client. Our switch-side approaches started by asking what kind of data we would want to send to the host and what our limitations were.

First, the data we wanted to send was real-time video as captured by our mounted cameras. Initially, this may seem as though our interests are the same as those of commercial video streaming services such as YouTube or Twitch. In fact, previous work has been done in the field of software-defined networking (SDN) applications by a group at the University of Würzburg, focusing on transferring YouTube video [3]. Similarly, a group at Stanford has released a paper on improving video transfer over wireless networks using techniques that affect multiple layers of the network[4].

However, real-time video streaming in those instances has had different meanings. In our instance, we are not so interested in providing a smooth, seemingly lossless video on the receiving end. We are more interested in acquiring the most recently captured frames, and in that sense we are essentially streaming images instead of video. On a similar note, we are also not interested in syncing or even streaming audio along with the video.

The only switches we can control between the host and the client are those between the host and the WAN. This is because we cannot control Wi-Fi or LTE connections on the wireless APs. We do, however, control the WiMAX tower, but there is not much support on other wireless APs for our software.

The idea we have for our switches is to give the video and data which we deem to be more important a greater portion of the available bandwidth. Even if we did have access to the Wi-Fi or LTE APs, the technology to control them using SDN is still in its infancy and is not implemented widely[1]. We are more interested in combining SDN solutions like those mentioned in Stanford’s original OpenRoads paper[5] which allows n-casting on multiple APs, with techniques that specifically target video transfer and take into account our control of the client and host.

Using the OpenFlow standard, we wrote a simple program to allow us to decide which clients get bandwidth priority in a network. We started out by simply dropping flows from non-priority streams. This presented the whole process as a sort of two-state system. In one state, all cameras are streaming to the processing host, and all streams are processed. During this state, the videos are all run through computer vision algorithms to determine which one is the most valuable to us.

Once a video is determined to be of highest priority, the system enters the second state. In this state, there is a smaller subset of higher quality streams going to the host. In our simple implementation, this is done on the switch using flow tables and packet dropping. In the future, however, we hope to implement this client-side, so as to decrease the networking load in the second state.

Lastly, we began to implement a simple message passing system in Python, in order to allow the video processing host to communicate with the switch controller. This will allow the host to tell the controller which clients are of lower priority and can be dropped.

If there are similarities between our switching and transferring ideas, this is not purely consequential. We can implement a virtual switch on the computer clients using Open vSwitch, which will allow us to run the switching code locally before the data is transmitted wirelessly. This means the switching code could act as client-side code as well, and further tests will have to be done to determine the best solutions.

V. Conclusion

After testing our actual network conditions, we found them to be scaled-down versions of the theoretical limits of the wireless technologies we hoped to use. We implemented a simple file transfer program to test data transfer while jumping between access points and interfaces. Lastly, we created a first attempt at limiting video traffic across the network switches using OpenFlow. The simplicity of each approach is stressed because we are already able to achieve what we set out to do; we hope, however, to see much more improvement on upload speeds and reliability by using the methods referenced throughout the paper.

We still have plenty of work ahead of us, but initial testing and prototyping have revealed problems we hope to solve and solutions we think we can implement. Hopefully, our work will ease some of the bigger problems with real-time video streaming in similar settings and allow further improvements in this field.

Acknowledgements

This research is supported in part by NSF grants CNS 1449860, CNS 1461932, CNS 1460971, CNS 1439672, CNS 1301774, ECCS 1231461, ECCS 1128209, and CNS 1138963.

References

[1] Manu Bansal, Jeffrey Mehlman, Sachin Katti, and Philip Levis. Openradio: a programmable wireless dataplane. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 109–114. ACM, 2012.

[2] Shyang-Lih Chang, Li-Shien Chen, Yun-Chung Chung, and Sei-Wan Chen. Automatic license plate recognition. *Intelligent Transportation Systems, IEEE Transactions on*, 5(1):42–53, 2004.

[3] Pouya Ostovari and Jie Wu. Robust wireless delivery of scalable videos using inter-layer network coding.

[4] Eric Setton, Taesang Yoo, Xiaoqing Zhu, Andrea Goldsmith, and Bernd Girod. Cross-layer design of ad hoc networks for real-time video streaming. *Wireless Communications, IEEE*, 12(4):59–65, 2005.

[5] Kok-Kiong Yap, Rob Sherwood, Masayoshi Kobayashi, Te-Yuan Huang, Michael Chan, Nikhil Handigol, Nick McKeown, and Guru Parulkar. Blueprint for introducing innovation into wireless mobile networks. In *Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures*, pages 25–32. ACM, 2010.