Performance evaluation of streaming algorithms for network cameras

Gonzalo Muñoz Ferrer *, Hugo Méric †, José Miguel Piquer ‡ and Javier Bustos-Jiménez ‡
*DCC - Universidad de Chile, Santiago, Chile
†INRIA Chile, Santiago, Chile
‡NIC Chile Research Labs, Santiago, Chile
Email: gmunoz@dcc.uchile.cl, hugo.meric@inria.cl, jpiquer@niclabs.cl, jbustos@niclabs.cl

***Abstract*—In this article we present the performance evaluation of different algorithms to distribute video frames from network cameras to multiple concurrent clients in real-time. The algorithms evaluated in this paper rely on a pool of buffers shared by all the clients. We implement these algorithms in the VLC media player and study their performance in terms of frame rate, hardware resource usage and decoding errors. Moreover, we compare their performance to the VLC streaming algorithm. Experimental results show that well designed algorithms combined with a sufficient number of buffers significantly outperform the VLC streaming algorithm.**

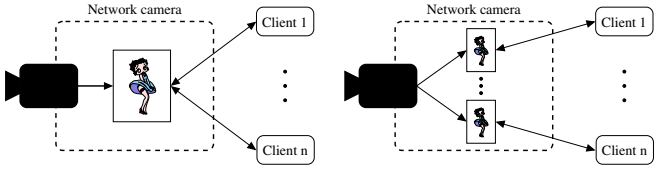
I. Introduction

Video over IP transmission has become popular since the advent of high-speed networks, good video encoding algorithms (reducing bandwidth usage while preserving the image quality [1], [2]) and very inexpensive video hardware; joining all of them in devices known as **network cameras**. The applications of network cameras include surveillance, positioning, target tracking, environment monitoring, etc. The usual scenario consists of multiple concurrent clients connected to such devices in order to receive the video stream in real-time.

Transmission of multimedia content over networks is an important research topic. Networks often suffer from packet losses and one of the biggest challenge of video delivery is to deal with these losses. To cope with packet losses, error resilience schemes (such as data partitioning and flexible macroblock ordering), error concealment and forward error correction are commonly used [1]–[3]. Also, another solution is to design congestion control algorithms adapted to real-time multimedia streaming [4]–[6]. Recently, cross-layer approaches are more and more used to deliver multimedia content as they obtain very good performance [7], [8].

Even if the literature concerning multimedia delivery is extensive [9]–[12], those works study this problem from the transport and/or end-user point of view, not from the one of the streaming server. Therefore, one can notice the problem of distributing video frames from a network camera to multiple clients is still an open research problem [13]. Our work has been focused on studying this challenging problem at the server side (i.e., the camera).

Several solutions can be used to deliver the video stream of a camera to several concurrent clients. The easiest solution is to store one frame in memory and then send it to all the clients as illustrated in Figure 1(a). However, a new frame can only be stored when all the clients have finished to read the current frame. This leads to poor performance as all the clients get the same frame rate determined by the client with the slowest connection. Another solution is to allocate one buffer for each client (see Figure 1(b)). Once a client has finished to read its buffer, the camera stores a new frame in its buffer. This solution provides each client with an optimal frame rate. Although this solution works with few clients, it is impossible to allocate one buffer per client in practical scenarios that involve hundreds of clients.



(a) One shared buffer (b) One buffer per client

Figure 1. Basic streaming solutions based on a pool of buffers for network cameras: (a) is the easiest solution with poor performance while (b) is the optimal solution that is not feasible in practical scenarios.

In practice, the camera has limited hardware resources and can only offer a small number of buffers. In that case, *the main issue is the performance, in terms of frame rate, of the clients with a fast connection*. Indeed, while clients with a slow connection read the available buffers, the camera cannot write new frames in these buffers.

In a previous work [14], we proposed different algorithms to manage a pool of buffers. The objective of these algorithms is to offer the best frame rate to clients with a fast connection even if many clients with a slow connection are connected to the camera. The frame rate performance was evaluated using Scilab [15] but no implementation in real hardware was done due to infrastructure limitations.

Nowadays, the scenario has been changed and several softwares already enable streaming in an easy-to-use (and also easy-to-study) way. For instance, the VLC media player is a well known open-source software for playing and streaming multimedia content [16]. The output stream of a network camera can serve as an input to VLC allowing to use the streaming algorithm implemented in the software. Moreover, VLC is an open source project that allows to develop its own algorithms inside the platform.

978-1-4799-3088-3/14/$31.00 ©2014 IEEE

In this paper, we evaluate and compare the VLC streaming solution with the algorithms developed in [14]. We implement these algorithms in VLC to obtain a fair comparison. We present the experimental results and compare the performance of both solutions in terms of frame rate, hardware resource usage and decoding errors. One of the most important result is that there is no decoding error for the clients with a slow connection when using the algorithms proposed in [14].

The paper is organised as follows: Section II introduces the two streaming solutions considered in this paper, the VLC implementation and algorithms based on a pool of buffers. Section III compares the performance of both solutions. Finally, we conclude and provide the future work in Section IV.

II. Network camera streaming algorithms

This section introduces the two solutions adopted to distribute video frames from a camera network to multiple clients.

A. VLC streaming algorithm

VLC already implements a streaming algorithm that relies on a HTTP server integrated in the platform. The HTTP server uses a single, fixed-size (5,000,000 bytes) circular buffer to store the data stream. The *HTTP module* writes the processed stream to the buffer, while a separate *host thread* reads data from the buffer and sends it to the clients. The *host thread* listens for new client connections and serves each connected client sequentially, allocating a separate buffer for each.

For each client, the *host thread* must copy data from the server buffer to the client buffer, if the latter is empty or if its content has already been sent to the client. Then the *host thread* performs a single *send* operation for the data in the client buffer that has not been sent yet, and moves on to the next client in the list.

The client connections are non-blocking, which means that multiple *send* operations can be required. In between these operations, the client's position in the server buffer could be overwritten if the client connection speed is too slow. In this case, the server updates the client's position to the latest data available and continues the process. Note that this phenomenon will cause decoding errors when the client decodes the stream.

B. Frame allocation and replacement algorithms

We quickly present the algorithms developed in [14]. More details can be found in the original paper.

The server involves two parts as shown in Figure 2: a *camera thread* reads the frames from the hardware and writes them into the pool of buffers, while a *client thread* picks a new frame from the pool and sends it to the corresponding client. There is one client thread per connected client and the number of buffers is generally much smaller than the number of customers. Also, the camera thread can only write a frame in a buffer that is not currently being read by a client because it will produce an uncompleted and invalid frame at the client side. Thus, the camera and client threads require to be synchronized to avoid simultaneous access [17].

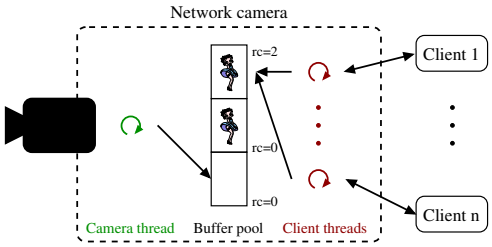


Figure 2. Network camera with a pool of (shared) buffers

When there is no free buffer (i.e., at least one client is connected to each buffer), the camera thread discards the current frame. Each buffer holds a frame with a *timestamp* of the time at which the frame has been generated and a *reference count* that indicates how many clients are using that buffer. For instance, two clients are reading the first buffer in Figure 2 (buffer at the top) so the reference count is equal to two. The timestamp and the reference count are used by the camera and client threads. Note that a free buffer is a buffer with a reference count equals to zero.

Frame replacement algorithm. The camera thread implements the frame replacement algorithm which chooses the buffer, among all the free buffers in the pool (i.e., buffers with a reference count equals to 0), to write the new frame. Three algorithms are considered:

- Oldest First (OF): the algorithm selects the buffer containing the frame with the oldest timestamp;
- Newest First (NF): the algorithm selects the buffer containing the frame with the newest timestamp;
- Any (ANY): the algorithm selects the first available buffer.

In [14], the authors observe that the frame replacement algorithm does not impact the frame rate of the client with a fast connection. Indeed, this algorithm affects which frames are in the pool but not the rate of new frames in the system.

Frame allocation algorithm. The client thread runs the frame allocation algorithm which selects the frame to send to the client. The frame allocation algorithm can only select a frame with a newer timestamp compared to the last frame transmitted to the client. Considering all the buffers that verify the previous condition, five algorithms are considered:

- Oldest First (OF): the algorithm selects the buffer containing the frame with the oldest timestamp;
- Newest First (NF): the algorithm selects the buffer containing the frame with the newest timestamp;
- Maximum Reference Count (MAX): the algorithm selects the frame contained in the buffer with the maximum reference count. Ties are resolved using the timestamp and picking the newest frame;
- Maximum Reference Count/Oldest First (MOF): the algorithm selects the frame contained in the buffer with the maximum reference count. Here ties are resolved using the timestamp and picking the oldest frame;
- Any (ANY): the algorithm selects the first usable frame.

The algorithms OF and NF only rely on the timestamps, while MAX and MOF use the timestamp and the reference count. ANY serves as a reference to evaluate the others algorithms.

Unlike the frame replacement algorithm, the allocation algorithm greatly impacts the performance in terms of frame rate. In Section III, we will see that this algorithm also affects significantly the CPU usage.

Implementation. We implemented the frame replacement and allocation algorithms into the HTTP module of VLC, which is an *output module*. Therefore, the HTTP module is at the end of VLC's content processing pipeline, and thus allows the algorithms to work with encoded video frames.

Two main changes are required in VLC for this implementation:

- Modify the structure for the memory allocated to the stream data on VLC's HTTP server, using a pool of frame buffers instead of a single circular buffer. This change effectively turns the HTTP module into the *camera thread* described in [14];
- Modify the HTTP *host thread* to spawn one *client thread* per client connected, instead of serving each connected client.

Since VLC's *host thread* serves every connected client, it cannot block while sending data. However, a *client thread* only sends data to the client it serves, and therefore establishes a *blocking* connection in the VLC implementation.

If a non-blocking connection is used, the thread might need to perform multiple *send* operations for the same data and this produces a steep rise in CPU usage, even when few clients are connected. A blocking connection ensures that each *client thread* is only active when it is needed.

III. Performance evaluation

We now present the performance evaluation of the algorithms and a comparison with the VLC streaming solution.

A. Simulations parameters

To evaluate the performance of each streaming algorithm, we use a video sequence with a resolution of 854x480 (Full Wide Video Graphics Array format). The video contains 14802 frames and is encoded at 29.97 Frame Per Second (FPS).

The encoding uses *Motion JPEG that compresses each frame separately*, which is a key point because it avoids any error propagation if a client skips some frames. Moreover, if the encoding is done using any motion compensation algorithm as in the H.264 format, this may incur error propagation if losses happen [1]. A part of future work will be to design the frame allocation and replacement algorithms when dependencies exist between the frames.

The aim of the algorithm developed in [14] is to offer the best frame rate to clients with a fast connection even if many clients with a slow connection are connected to the camera. Thus our simulations involve two clients with a fast connection with up to 260 clients with a slow connection. The connection speed of slow clients is in the range of 149 - 151 KiB/s, while fast connections are equal to the *localhost speed*, enabling the clients to read all the transmitted frames.

At the beginning of each simulation, the clients connect to the server. Once all the clients are connected, the performance measurement begins. The frame rate is computed by dividing how many different frames are displayed during the measurement by the simulation time. In order to get the CPU and physical memory usage, we use the command `top` [18].

All the simulations runs on a *single computer* Dell Inspiron N4020 with a processor Intel Pentium T4500 at 2.3 GHz. Thus the computer simulates both the server and client parts. The operating system is Ubuntu 11.10. Table I resumes the simulations parameters.

Table I
SIMULATIONS PARAMETERS

Encoding format	Motion JPEG
Encoding speed	29.97 FPS
Video resolution	854x480
Number of fast connection clients	2
Fast connection speed	localhost speed
Number of slow connection clients	0 - 260
Slow connection speed	149 - 151 KiB/s

B. Frame rate for fast connection clients

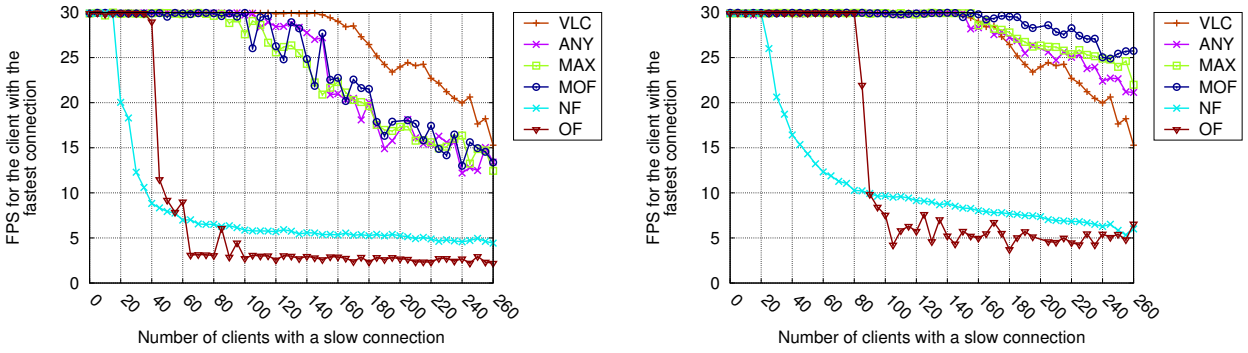
We present the performance in terms of frame rate for one of the clients with the fastest connection in order to study how increasing the number of clients will reduce the performance of fast connection clients. We do not present the results for the slow connection clients as the aim is to maximise the frame rate for fast connection clients under heavy load.

First, note that we set the replacement algorithm to NF for all the simulations in the rest of the paper, because we observed that the frame replacement algorithm does not affect the frame rate, which is consistent with the results in [14].

Figure 3 shows the performance of the frame allocation algorithms with 8 and 12 buffers. The experimental results differ slightly from the simulations in [14]. In practice, the algorithms MAX, MOF and ANY perform similarly while the Scilab results exhibit better performance for the algorithms based on the reference count. This result is very interesting as the algorithm ANY is very easy to implement.

Another difference is the slope of the algorithm OF. In the simulations, the transition from good (30 FPS) to bad (5 FPS) performance is very fast.

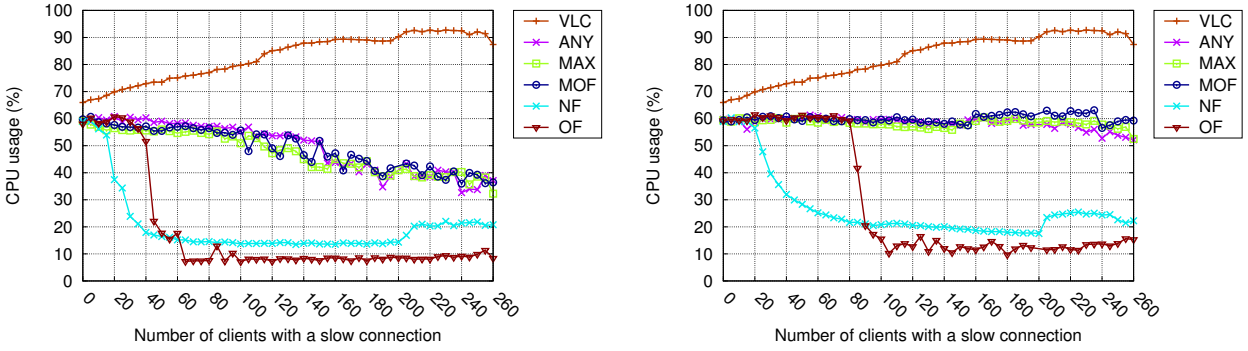
Also, Figure 3 shows that algorithms based on a pool of buffers are more effective than the VLC streaming algorithm if the number of buffers is sufficient. Indeed, when there are more than 10 buffers, experimental results show that the MAX, MOF and ANY algorithms outperform the VLC implementation under heavy load. For instance, considering the case with 250 slow connection clients connected to the network camera, the VLC algorithm offers a frame rate (for the fast connection client) of 18 FPS against 24 FPS for the MAX algorithm if 12 buffers are available (see Figure 3(b)). This represents an improvement of 50%.



(a) Buffer pool with 8 buffers

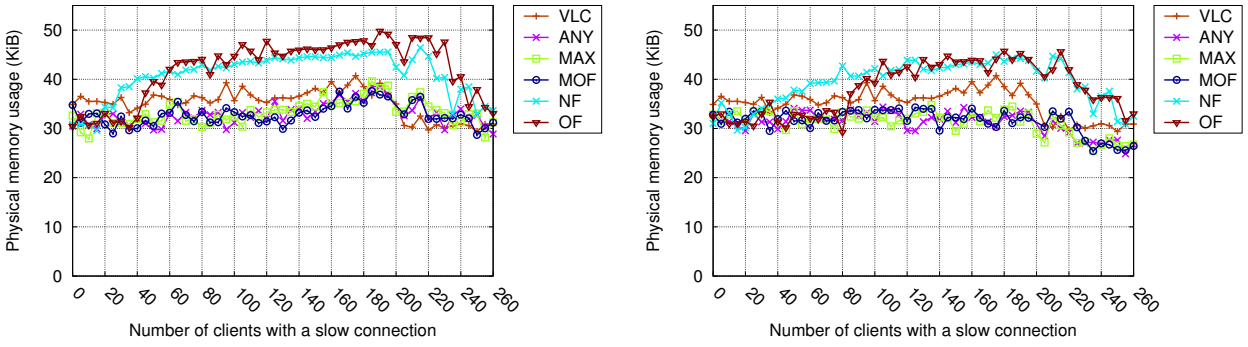
(b) Buffer pool with 12 buffers

Figure 3. Comparison of the frame allocation algorithms in terms of FPS for the client with the fastest connection. For the simulations, we set the frame replacement algorithm to NF and we only vary the frame allocation algorithm.



(a) CPU usage (8 buffers)

(b) CPU usage (12 buffers)



(c) Physical memory usage (8 buffers)

(d) Physical memory usage (12 buffers)

Figure 4. Comparison of the hardware resource usage (CPU and physical memory) for VLC and the algorithms proposed in [14]. For the simulations, we set the frame replacement algorithm to NF and we only vary the frame allocation algorithm.

C. Hardware resource usage

The next results present the CPU and physical memory usage given by the command `top`.

CPU. Figure 4(a) and Figure 4(b) present the CPU usage (i.e., the amount of time used by the processor to complete the task) at the server side (the camera) when streaming with VLC and the different frame allocation algorithms.

The results point out that the algorithms based on a pool of buffers always use less CPU than the VLC algorithm, even when the performance in terms of frame rate are better. For instance, if 250 slow connection clients are connected, VLC offers 18 FPS to the client with the fastest connection and used 90% of the CPU. However, the algorithms MAX, MOF and ANY combined with 12 buffers give a frame rate of 24 FPS (Figure 3(b)) while the CPU usage is around 60% (Figure 4(b)).

We also observe that the CPU usage decreases when the number of clients with a slow connection increases. This can be explained as the probability to find a free buffer decreases with the number of slow connection clients. In the case where no free buffer is available, the camera thread sleeps until a buffer gets free and thus the task uses less CPU time.

This observation also explains why the frame rate (Figure 3) and CPU usage (Figure 4(a) and Figure 4(b)) curves are similar. Indeed, the algorithms MAX, MOF and ANY obtain the same performance in terms of frame rate and CPU, while the algorithms NF and OF use less CPU but also present worse performance in terms of frame rate. For the algorithm OF combined with 12 buffers, we observe in Figure 3(b) that the frame rate decreases quickly when 90 slow connection clients are connected, which corresponds to the steep CPU decrease in Figure 4(b).

Physical memory. In Figure 4(c) and Figure 4(d), we propose to compare the physical memory usage of the different streaming solutions. The physical memory corresponds to the column RES when running the command `top`.

For the algorithms proposed in [14], the simulations show that the memory usage does not vary much, but the tendency is to use less memory when the FPS grows. For instance, the algorithms MAX, MOF and ANY use less memory than NF and OF in Figure 4(d). Moreover, when the number of buffers increases, the frame rate is larger and we can see that the memory usage is also lower.

This last point is however not intuitive as it seems natural to use more memory if there are more buffers. This phenomenon may be a consequence of the memory management that the operating system does to handle the server and the clients (each a separate VLC process) on the same machine.

Compared to VLC, the performance in terms of physical memory usage is similar and we do not have significant improvements as with the frame rate and the CPU.

D. Decoding errors for slow connection clients

The final results concern the decoding errors that happen for slow connection clients. As already explained in Section II, the client's position in the server buffer could be overwritten if the client connection speed is too slow causing decoding errors at the client side. The goal of this part is to quantify the number of errors and their impact on the decoded stream.

Simulations setup. The following simulations only involve one slow connection client connected to the server. The client has a connection speed of 150 KiB/s. We propose to compare the VLC algorithm with the frame allocation algorithm MAX (combined with the frame replacement NF), with 8 buffers available at the server.

To obtain the decoding errors, we modify VLC's *decode module* to detect the errors reported by the decoding functions that are normally presented as *debug* messages during VLC's execution. Thus we can count the errors and output the results at the end of each simulation.

Results. For each algorithm we run 10 simulations of 5 minutes. Table II resumes the average results.

For our implementation of the replacement and allocation algorithms, there is no decoding error and the FPS at the server and at the client side are similar.

With VLC, the results show that many decoding errors happen and also the frame rate at the client is much lower than at the camera.

Table II
DECODING ERRORS AND FPS STATISTICS FOR ONE CLIENT WITH A SLOW CONNECTION

	VLC algorithm	Algorithms in [14]
Decoding errors	149.8	0
Errors per minute	29.27	0
FPS at the client	0.72	1.47
FPS at the server	1.24	1.49

As explained in Section II, the decoding errors are a consequence of overwriting the client's position in the server buffer. More specifically, while the *host thread* sends data from the client buffer to the client, the *HTTP module* continues to write new video data to the HTTP server buffer.

Nevertheless, the client buffer does not always contain complete frames. So when, for example, half a frame is copied to the client buffer, the other half in the server buffer could be overwritten by new data **before** the client buffer's content has been sent to the client.

This results in the client receiving only half of a frame, which VLC cannot decode correctly. When this happens, VLC discards the incomplete frame and the video seems to be frozen on the last correctly decoded frame until a new decodable frame is received. This explains why the camera and client frame rates are different.

It is important to note here that the VLC streaming algorithm will produce this type of decoding errors even when there is no packet loss during data transmission from server to client: slow clients will receive incomplete frames even when using reliable connections. In contrast, this does not happen when using the frame allocation and replacement algorithms.

IV. CONCLUSION AND FUTURE WORK

In this paper, we presented the performance evaluation of various streaming algorithms developed for the server side of network cameras. Based on an implementation in VLC, we study the performance of these algorithms in terms of frame rate, hardware resource usage and decoding errors. The experimental results point out that streaming strategies based on a pool of buffer (as introduced in [14]) may outperform the VLC streaming algorithm. Indeed, if there are enough buffers, clients with a fast connection have a best frame rate in scenarios under heavy load. The CPU and physical memory usage are also improved compared to VLC. Finally, there is no decoding error for the clients with a slow connection which ensures that the stream can be played correctly.

In a future work, we plan to to develop an encoding method with motion compensation such as H.264/AVC [1], in order to evaluate our algorithms against the top-of-the-line used standards. We also plan to investigate the impact of the streaming algorithms on the quality of experience perceived by the clients. Indeed, we noted in [14] that there is a tradeoff between the frame rate and the *jitter* of the transmission, and it has been demonstrated that this parameter is highly related with the users quality perception [19], [20].

REFERENCES

[1] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, no. 7, pp. 560–576, 2003.

[2] S. Wenger, "H.264/AVC over IP," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, no. 7, pp. 645–656, 2003.

[3] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *SIGCOMM Computer Communication Review*, vol. 27, no. 2, pp. 24–36, 1997.

[4] J.-C. Bolot and T. Turletti, "Experience with control mechanisms for packet video in the internet," *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 1, pp. 4–15, 1998.

[5] R. Rejaie, M. Handley, and D. Estrin, "Rap: An end-to-end rate-based congestion control mechanism for realtime streams in the internet," in *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 1999, pp. 1337–1345.

[6] H. Kanakia, P. P. Mishra, and A. R. Reibman, "An adaptive congestion control scheme for real time packet video transport," *Networking, IEEE/ACM Transactions on*, vol. 3, no. 6, pp. 671–682, 1995.

[7] D. Pradas, A. Bouabdallah, J. Lacan, M. Vazquez Castro, and M. Bousquet, "Cross-layer optimization of unequal protected layered video over hierarchical modulation," in *Global Telecommunications Conference, IEEE*, 2009, pp. 1–6.

[8] G. Sarwar, R. Boreli, and E. Lochin, "Xstream-X264: Real-time H.264 streaming with cross-layer integration," in *Multimedia and Expo (ICME), IEEE International Conference on*, 2011, pp. 1–4.

[9] Y. Liu, Y. Guo, and C. Liang, "A survey on peer-to-peer video streaming systems," *Peer-to-peer Networking and Applications*, vol. 1, no. 1, pp. 18–28, 2008.

[10] B. Vandalore, W.-c. Feng, R. Jain, and S. Fahmy, "A survey of application layer techniques for adaptive streaming of multimedia," *Real-Time Imaging*, vol. 7, no. 3, pp. 221–235, 2001.

[11] S. Misra, M. Reisslein, and G. Xue, "A survey of multimedia streaming in wireless sensor networks," *Communications Surveys & Tutorials, IEEE*, vol. 10, no. 4, pp. 18–39, 2008.

[12] S. Thombre, G. Diyewar, S. Barai, and P. Tiwari, "Online video streaming system." *International Journal of Advanced Research in Computer Science*, vol. 4, no. 2, 2013.

[13] M. Lindeberg, S. Kristiansen, T. Plagemann, and V. Goebel, "Challenges and techniques for video streaming over mobile ad hoc networks," *Multimedia Systems*, vol. 17, no. 1, pp. 51–82, 2011.

[14] J. M. Piquer and J. Bustos-Jiménez, "Frame allocation algorithms for multi-threaded network cameras," in *Proceedings of the 16th international Euro-Par conference on Parallel processing*, 2010, pp. 560–571.

[15] B. Wu and A. Bogaerts, "Scilab-a simulation environment for the scalable coherent interface," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1995. MASCOTS'95., Proceedings of the Third International Workshop on*. IEEE, 1995, pp. 242–247.

[16] VLC Media Player. [Online]. Available: https://wiki.videolan.org/

[17] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley & Sons Publishing Co., 2013.

[18] Ubuntu Manpage. [Online]. Available: http://manpages.ubuntu.com/

[19] R. Steinmetz, "Human perception of jitter and media synchronization," *Selected Areas in Communications, IEEE Journal on*, vol. 14, no. 1, pp. 61–72, 1996.

[20] M. Claypool and J. Tanner, "The effects of jitter on the peceptual quality of video," in *Proceedings of the seventh ACM international conference on Multimedia (Part 2)*. ACM, 1999, pp. 115–118.