Frame Allocation Algorithms for Multi-threaded Network Cameras

José Miguel Piquer[1] and Javier Bustos-Jiménez[2]

[1] Departamento de Ciencias de la Computacion (DCC), Universidad de Chile
jpiquer@dcc.uchile.cl
[2] Escuela de Ingeniería Informática, Universidad Diego Portales
javier.bustos@mail.udp.cl

Abstract. This paper presents a first attempt to solve a challenging problem, proposing novel and successful algorithms to efficiently distribute video frames from network cameras to many concurrent clients.

The usual scenario studied is composed of a camera generating video frames at a given rate and distributing them over a network to several concurrent clients. In general, the idea is to allocate one thread per client at the camera, sharing a pool of one-frame buffers. The algorithms studied consider the allocation of buffers to new frames and the allocation of frames to clients.

We study different combinations of algorithms, buffers and clients in order to find an optimal solution for the usual scenarios we face when the network camera is under heavy use. The main conclusion is that frame allocation algorithms have a strong impact on system performance: under the same conditions, client performance improves from 4 to 25 frames per second with the best algorithm combination at the camera.

Keywords: Video over IP, frame allocation, network camera.

1 Introduction

The Video over IP transmission is becoming popular with the advent of high-speed networking, good video encoding algorithms (reducing bandwidth usage while preserving the image quality [12,13]), and the massification of inexpensive video hardware. This has created a lot of new challenges for the Internet, particularly related to congestion control [8].

Also, a lot of research has been done on video compression [4,5], transmission with packet loss recovery [10,2] or using Quality of Service extensions [9,6]. This has enabled the advent of *network cameras*: digital video cameras integrated with a network connection, accepting client connections from the Internet and sending them a real-time video stream.

A typical network camera includes a web server to authenticate users and a video server to send the video frames to the clients using a proprietary protocol (running over UDP or TCP). A network camera is expected to support concurrent clients with limited hardware resources, namely memory and processor. A typical commercial Axis camera limits the number of clients to a maximum of 20. Our objective is to develop

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 560–571, 2010.

algorithms to support more than 100 concurrent clients with less memory than a typical Axis camera, using only 500 Kbytes of RAM. The configuration used is with color 640x480 resolution frames, JPEG encoded giving usual sizes of around 50 Kbytes each. The main goal is to support as many clients as possible with a given hardware capacity.

Another important objective of our algorithms is to be able to deliver 30 frames per second to a fast client in a high-speed network. So, we propose algorithms preserving that performance for fast clients even if many clients are much slower than that. As an example, a home user connected at 256 Kbps will be able to receive only one frame every two seconds whereas a local 100 Mbps ethernet user will be able to receive 30 frames per second using almost 10 Mbps bandwidth. To provide this service, the camera needs to store video frames in its own memory while they are being transmitted to their clients. However, we cannot afford to allocate one frame per client, as memory is a scarce resource in the camera.

The problem studied in this paper is how to manage a bounded pool of frames in memory, choosing which frames to keep and which frames to send to the different clients running at different speeds. This problem, to our knowledge, has not been studied previously in the literature.

In our experiments, we send a complete frame to each client, using Motion JPEG [1] as the video codec. If a differential encoder is used (as H.261 [4]), the frame allocation problem remains the same: we need to choose a frame to calculate the differences with the last one, and to keep a pool of frames to choose from. However, we have not tested differential encoders, and all the experiments performed for this paper were carried out sending complete JPEG video frames.

The paper studies the server side of the problem (camera). Of course, the client can provide some buffering or even to store the whole video to play it afterwards. However, if we want to stream the video from the camera in real time, buffering does not help to solve the different capacities between clients, it only helps to conceal variance in point to point bandwidth. In this scenario, only the server can solve the problem.

The main contributions of this paper are: a comprehensive study of a new area of research for network cameras, a list of new algorithms to manage frames and clients, and a performance analysis based on simulation, and a proposal of efficient combinations of algorithms for frame allocation.

The paper is organized as follows: Section 2 presents the problem and the algorithms involved. Section 3 explains each algorithm in detail. Section 4 presents the designed experiments and the obtained results. Finally, Section 5 presents our conclusions.

2 Frame Allocation and Replacement

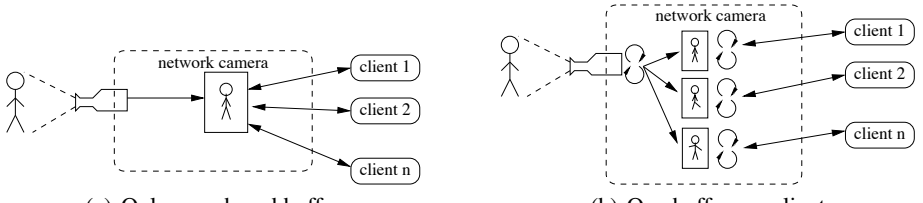
A network camera with only one client is always running an infinite loop getting the next frame from the hardware, copying it into memory, and then sending it to the client over the network. The main performance measures considered are the frames per second (FPS) that the camera hardware can generate and the FPS that can be displayed at the client. The FPS depends on several factors, such as frame size, resolution, color depth, network bandwidth, protocol used, etc. In this paper we concentrate only on the parameters that are directly controlled by the camera itself and leave out the network protocol used.

If a network camera can generate a maximum of x FPS, and we have a client that can display up to y FPS, we suppose that the network camera with *only* this client is always able to deliver the $min(x, y)$ FPS at the client. We call a client that is able to display at the camera FPS or higher ($y \geq x$), a *Fast Client*. We define as *Slow Client* a client that is only able to display at an FPS five times lower than the camera FPS ($5y \leq x$). In a real environment, it means a client receiving at less than 6 FPS.

2.1 Concurrent Clients

When a network camera has more than one connected client, it must send the same video stream over different connections. The simplest algorithm is to read the next video frame into memory and then send it to every client. However, in this case, a new frame cannot be read into memory (and thus must be discarded) until the slowest client has finished receiving the previous one. Thus, all clients are forced to receive at the same frame rate, determined by the slowest client. In practice, all clients perform at the slowest client frame rate (Figure 1(a)).

If we denote by y_i the FPS of the client i, every client is receiving the video at $min(x, y_k)$ FPS, where $y_k = min(y_i), \forall i$.



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

Fig. 1. Camera buffering

A better solution is to develop a multi-threaded server, with one thread per client, a thread for the camera, and a separate buffer allocated in memory for each client (see Figure 1(b)). The camera thread must copy each new frame to all free client buffers. This solution provides each client with an optimal frame rate, as we have every client i receiving the video at $min(x, y_i)$ FPS. This is the optimal solution, as it is equivalent to the case where each client is connected alone to the camera, as long as the camera is able to copy the frames to all the buffers without degrading its own FPS. However, memory is a scarce resource on the camera and in practice we are forced to limit the number of buffers in memory to a fixed maximum number to ensure that we do not run out of memory. On the other hand, we do not want to limit the number of clients to the number of buffers in memory. In general, we want to be able to support a number of clients much higher than the number of buffers (around 10 times), without degrading the overall performance too much.

2.2 The Frame Pool

We designed a solution that is in between the two previous algorithms: sharing a pool of frames between clients to have more than one frame allocated in memory but less than one frame per client.

To enable frame sharing, the server is divided in two parts: a *camera thread* reading the frames from the hardware and putting them into a frame pool, and several *client threads* (one per client) retrieving a frame from the frame pool and sending it to the corresponding client (see Figure 2).

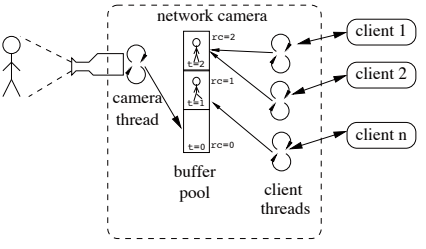


Fig. 2. Many shared buffers

The frame pool is implemented using a fixed-length array of buffers (depending on the camera memory size), so the *camera thread* and the *client threads* must synchronize between them to block when the buffer pool is either full or empty.

This synchronization problem is very similar to the well-known readers/writers problem [11], but with a few stronger restrictions[1]:

Each buffer holds a frame with a timestamp of the time at which the hardware generated it and a reference count which is a counter of the number of *client threads* currently using that buffer (see Figure 2).

Each time the *camera thread* gets a new frame from the hardware, it looks for an available buffer where to write the frame. In general, all buffers are holding a frame, so one must be selected to be replaced by the new one. A buffer that has one or more *client threads* reading it (i.e. has a reference count greater than zero) cannot be replaced. So, the only candidates for replacement are the frames with reference count zero. If there is none, the *camera thread* blocks until such a frame is available.

Each time the *client thread* needs a new frame to send to its client, it looks for a buffer holding a usable frame. A usable frame is any of the frames in buffers that have a timestamp newer than the last frame it sent to its client (to ensure the monotony of time for the video). In pseudo-code, the *camera thread* and the *client threads* can be seen in Figure 3.

Where camera frame replacement (`get_free_buf()`) is the algorithm executed by the *camera thread* to select a buffer to hold a new frame received from the hardware. As

[1] This problem is a very good example of process synchronization for an Operating Systems or Concurrent Programming course. We have been using it successfully: it is complex yet understandable.

```
camera_thread() {
  for(;;) {
    buf = get_free_buf();
    buf.frame = read_camera();
    buf.timestamp = time();
    put_buf(buf);
  }
}
```

```
client_thread(socket) {
  t=0;
  for(;;) {
    buf = get_next_buf(t);
    send(buf.frame, socket);
    t = buf.timestamp;
    free_buf(buf);
} }
```

Fig. 3. Pseudo-code for camera and client threads

all the buffers hold a frame in steady state, a frame will be deleted by this operation. The frame replacement algorithm is implemented by `get_free_buf()`, a function that finds buffers with a reference count zero and selects the best candidate discriminating by the timestamp value. If no buffer with reference count zero exists, the function blocks until one is available (signaled by `free_buf()`).

And, client frame allocation (`get_next_buf()`) is the algorithm executed by all the *client threads* each time they have finished sending a frame to select the new frame to transmit. The client frame allocation algorithm is implemented by `get_next_buf()`, a function that finds a buffer with a timestamp greater than the last one sent by the *client thread*. If many buffers with suitable timestamps are available, it can use the reference counts and the timestamps to choose one. If none is available, the function blocks until a new frame is generated by the *camera thread* (signaled by `put_buf()`). Different *client threads* can share the same buffer, as they are only reading it. The reference counts indicate how many *client threads* are using each buffer; `put_buf()` initializes the reference count of the buffer to zero, `get_next_buf()` increments it and `free_buf()` decrements it.

While the *camera thread* is blocked waiting for a buffer where to put the next frame, the system is losing frames from the hardware which continues to capture at the hardware FPS (x) but no thread is reading. All performance differences on the client side are related to this effect: if the camera blocks for more than $1/x$ seconds, the FPS at fast clients immediately drops. If the camera never blocks, fast clients receive the maximum FPS. Therefore, all good algorithms improve the probability of finding a free buffer to put a new frame to avoid the blocking of the *camera thread*.

3 Proposed Algorithms

We designed three frame replacement and five frame allocation algorithms that can be combined freely. We implemented all of them in order to compare their final performance.

3.1 Camera Frame Replacement

We studied three different algorithms for the *camera thread* to choose a frame to be replaced, given a list of all the frames stored in buffers with reference count zero.

Oldest First (OF): Selects the frame with the oldest timestamp. This algorithm replaces the frames one by one, erasing always the oldest frame in the pool, supposing it is the less useful for the clients.

Newest First (NF): Selects the frame with the most recent timestamp. This algorithm tends to renew the last frame generated if only slow clients are present, thus using only a few buffers and minimizing the number of useful frames in memory for every client.

Any Frame (ANY): Selects the first available frame. This algorithm searches sequentially the buffers and selects the first frame that has reference count zero. Hence, it chooses a frame essentially arbitrarily and serves mainly as a benchmark to test if using the timestamp information improves the performance.

3.2 Client Frame Allocation

We studied five different algorithms for the *client threads* to choose the next frame to be sent, given the list of all the frames stored in buffers with timestamps greater than the last frame they sent.

Oldest First (OF): Sends the frame with the oldest timestamp. This algorithm sends the frames in order without skipping frames that are still available in memory.

Newest First (NF): Sends the frame with the most recent timestamp. This algorithm sends the new frames as quickly as possible, skipping all the intermediate frames and catching up with the newest available.

Maximum Reference Count (MAX): Sends the frame with the maximum reference count. This should increment the number of available buffers with reference count zero at any given time. Ties are resolved using the timestamp, picking the newest. This algorithm improves buffer sharing, maximizing reference counts.

Maximum Reference Count/Oldest First (MOF): Sends the frame with the maximum reference count. Ties are resolved using the timestamp, picking the oldest.

Any Frame (ANY): Sends an arbitrary usable frame. This algorithm provides an arbitrary behavior. As before, it serves mainly as a baseline to compare with the others.

4 Experiments and Results

We carried out two parallel experiments: one was executed using a real network and many clients connecting to a camera server and the other was a simulation ran on the scientific simulator Scilab. The first experiment was run mainly to ensure that our simulated results were consistent with the real ones. The Internet is a complex system to be simulated [3,7], so it is important to verify our results. The real experiment was deployed up to 15 clients and the simulation up to 100. We will only show here the results from the simulation, as the real experiment was fully consistent with it.

For this work, we do not consider the network and protocol behaviors, and we focus only on the efficient management of the frames in the camera memory.

4.1 Frames per Second

We first tested all combinations of algorithms using one main measure: frames per second obtained by a Fast Client (see section 2), while we keep adding Slow Clients to the camera. We designed a realistic scenario for the experiment: we fixed the number of buffers allocated in memory, we ran two Fast Clients permanently, and we added Slow Clients one by one, measuring the FPS received by the fastest client.

The two Fast Clients were capable of receiving 34 FPS and 24 FPS respectively. The Slow Clients were able to display between 3.5 and 5.5 FPS. To avoid synchronization between the arrivals of clients requests, we forced all the clients to have different speeds. Whenever asked by `read_camera()`, the camera always returns a new frame, so there is no limitation on the FPS from the camera itself.

The fastest client FPS depends only on the FPS that the *camera thread* is able to put into the buffer pool: a *client thread* will only block waiting for a frame if no frame newer than the last it has sent is available, and this only happens if the *camera thread* was not able to put that frame in the pool (the only exception would be if the client is faster than the camera, but this is not the case here).

Frame Allocation. To select the best Frame Allocation algorithms, we run each experiment during 10 minutes, generating 20,000 frames at the camera. The results for 4, 6 and 8 buffers fixing Newest First as the Frame Replacement algorithm, are shown in Figure 4. Each point in each graph is the mean FPS measured in the fastest client during 20,000 frames generated at the camera.

For the frame allocation algorithm, we can separate three groups from the graphs: the best group (MAX and MOF), the intermediate group (ANY and NF), and the bad one (OF). Note that with 100 slow clients and only 8 buffers, the best group can still keep the fastest client at 25 FPS while the worst has it already at 5 FPS (actually this is equivalent to the slowest client we have). MAX performs better than MOF at the end. During the very early stages of the test, with only a few clients, MOF tends to be a little better. Anyhow, MAX is clearly better at high loads, when performance is crucial. OF is interesting because it behaves very well at the beginning but degrades fast and finishes even worst than ANY.

We can conclude that the best algorithm (MAX) allows us to serve 100 clients in our experiment with 8 buffers, degrading the fastest client only from 34 to 26 FPS. So we are able to serve 12 clients per buffer with a degradation of only 24% in the frame rate. If we look at the 6-buffer case, we are able to serve 45 clients with 25 FPS, giving 7 clients per buffer. In the 4-buffer case, we get 12 clients with 25 FPS, giving us 3 clients per buffer. Even though these are preliminary results, it seems that for MAX, the number of clients increases better than linear with respect to the number of buffers. We ran again all these expermients with different Frame Replacement algorithms fixed as default (to see if the use of NF played a role in the results) and the curves were exactly the same.

Frame Replacement. To select the best Frame Replacement algorithm, we re-ran each experiment for 10 minutes, generating 20,000 frames at the camera. The results for 4, 6 and 8 buffers using MAX as the Frame Allocation algorithm are shown in Figure 5. Each point in the graph is the mean FPS measured in the fastest client during 20,000 frames generated at the camera.

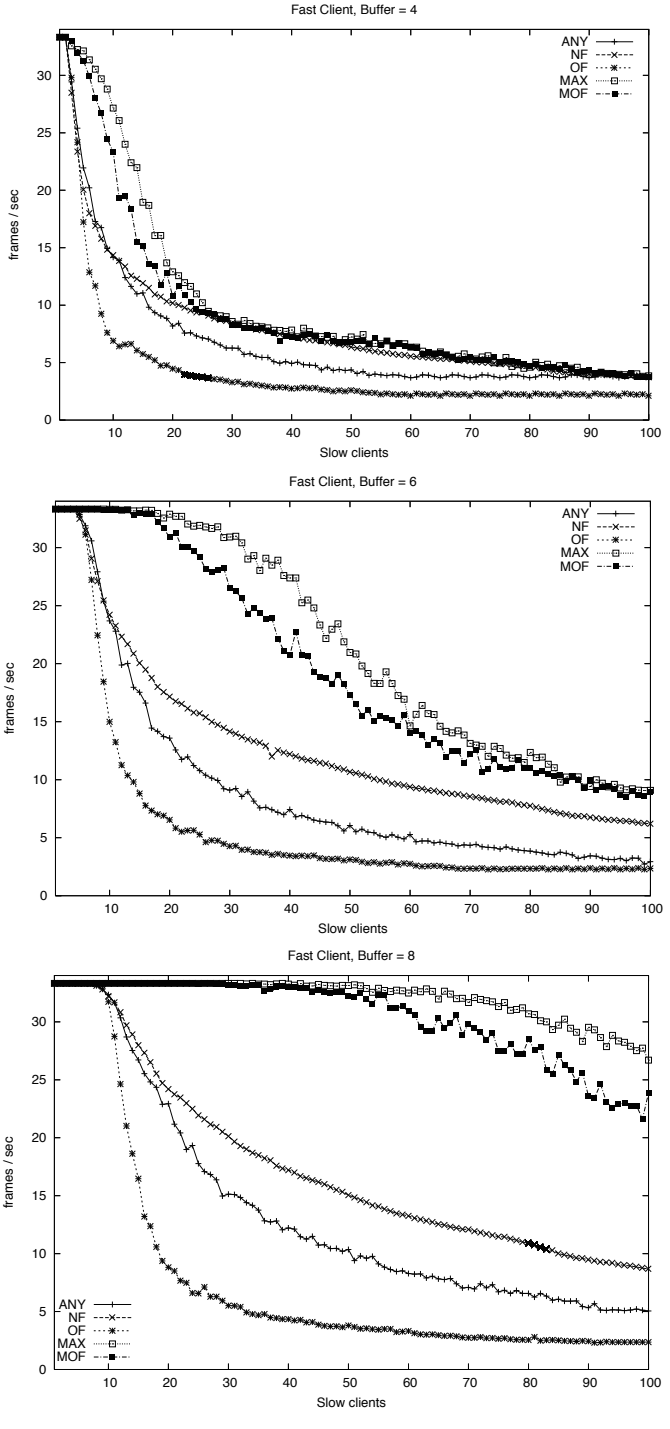


Fig. 4. Frame Allocation and FPS, 4,6 and 8 buffers

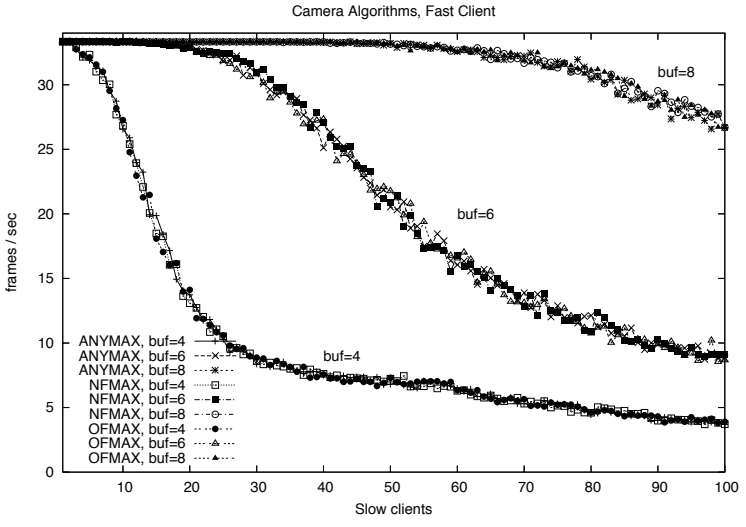


Fig. 5. Frame Replacement FPS, 4, 6, 8 buffers

Note that once the number of buffers and the Camera Frame Allocation algorithm are fixed, the Frame Replacement algorithm is irrelevant with respect to the FPS received at the fastest client. This can be expected, as different algorithms do not affect the FPS put into the buffer pool by the camera, because they only replace different frames. This behaviour is confirmed by measures taken with all the other Camera Frame Allocation algorithms.

4.2 Distance between Frames

We noted in previous experiments that Frame Replacement algorithms do not affect the FPS received by the fastest client. However, the algorithms do select different frames to replace, so the clients are getting the same FPS but they are not receiving the same frames. This difference may be perceived by the user, as some algorithms generate jumps in the frame sequence and at certain point this could become annoying. To study this effect, we measured the perceived smoothness of the video stream.

The smoothness of a video stream can be measured calculating the distance (number of skipped frames) between consecutive frames received at a given client. If the frames generated at the camera are numbered sequentially, each client knows exactly how many frames is skipping each time it receives a new frame. We call this number *frame distance*.

As the FPS is the same for all algorithms, the mean frame distance will also be the same in the long run. However, the standard deviation of the frame distance can be different depending on the algorithm combination. The idea is that a small deviation should be better than a big deviation, as a sequence of equally spaced frames is better than a sequence where some frames are very near in time and then a big skip occurs. We

suspected that the camera replacement algorithms could make a difference here, as they replace different frames in the buffer pool. Also, we suspected that the reference-count based client algorithms (MAX and MOF) could be worse in terms of distance variation as they select the frames to send primarily based on their reference count instead of their timestamps.

To measure the distance variation, we picked one Slow Client in the same experiments, measuring its frame distance variation using 8 buffers. The Fast Clients are not very interesting as they are able to receive the full stream of frames, so the frame distance is too low. We measured with 8 buffers, as a high number of buffers generates more options for the algorithms and thus should generate more variation.

Frame Allocation Algorithms. Figure 6 shows, for the NF Frame Replacement algorithm, combined with the five Frame Allocation algorithms, the standard deviation of frame distance divided by the mean distance, to normalize it to be between 0 and 1. In this case, a lower value is better than a high value: 0 corresponds to a constant stream without variance on the frame distance, while a value 1 indicates that we cannot predict the distance of the next frame based on history, i.e. a very variable stream of frames.

Note that the different algorithm combinations generate very different results in terms of frame distances. As expected, ANY is the worst and the reference-based algorithms are worse than the time-based algorithms. It should be noted than a variance near 1 means that there is almost no correlation between a pair of frames received in sequence. How disturbing is a level of variance around 0.5 to a human user is something to be studied in the future. If it is a serious problem, maybe a tradeoff between FPS and frame variance will be needed.

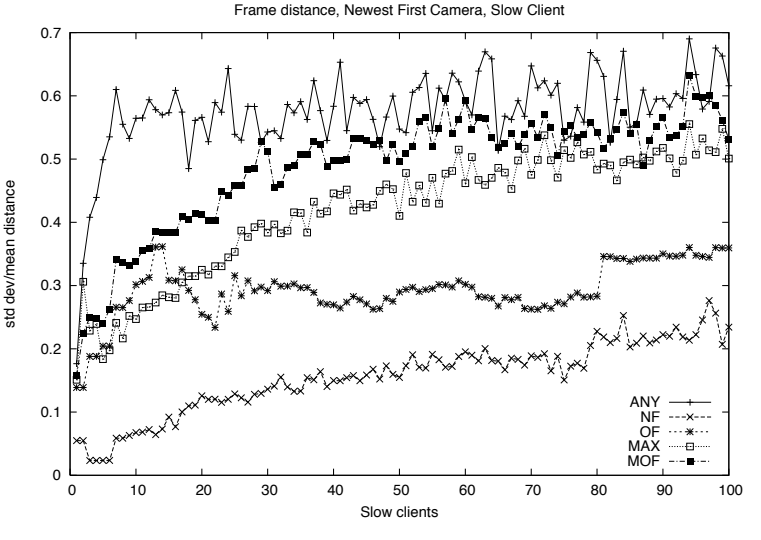


Fig. 6. Frame Allocation and Variance

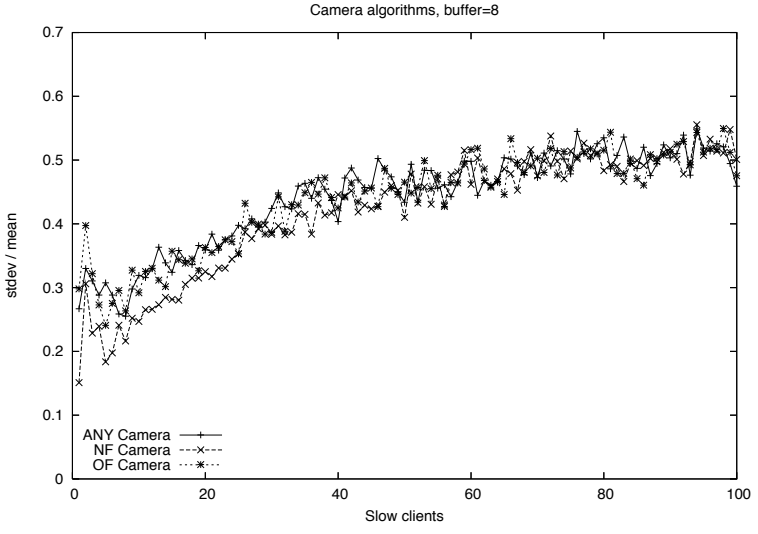


Fig. 7. Frame Replacement and Variance

Frame Replacement. To see if the Frame Replacement algorithms play a role in the variance between frames, we fixed the Frame Allocation to MAX (the best algorithm in terms of FPS) and we compared the obtained variance between the three Frame Replacement algorithms (see Figure 7).

Note that in the first part of the graph, NF generates a lower variance. After 25 clients, the algorithms are equivalent.

5 Conclusions

In this paper we presented a complete study of different algorithms to distribute video to many concurrent point-to-point clients with a limited number of buffers in memory. The different algorithms tested showed that the best choice for the client Frame Allocation algorithm, in terms of FPS, is the Maximum Reference Count (MAX algorithm). However, MAX generates high variance in the frame distance, although using Newest First (NF) as the camera Frame Replacement algorithm we can alleviate this in the first stages. This solution can handle 12 clients per buffer with 8 buffers, with a degradation of only a 24% for the fastest client (from 34 to 26 FPS) and a frame distance standard deviation over the mean of 0.5 (around +/- 10 frames).

If FPS maximization is not the main goal and the frame variance is the most important measure, the combination NF/NF seems to be the best choice, but at an enormous cost in FPS: with 12 clients per buffer and 8 buffers, we get a degradation of 74% for the fastest client (from 34 to 9 FPS) and a frame distance standard deviation over the mean of 0.2 (around +/- 1 frame).

In our experiments, we increased the numbers of buffers from 4 to 8 and the performance of the MAX algorithm gave that the number of clients increased better than

linear with respect to the number of buffers. As memory will continue to be an issue in the future on all network cameras, as the number of clients keeps growing, the solutions proposed here are crucial to limit the memory consumption in the camera.

A lot of future work remains to be done: to study the curves beyond 100 clients and 8 buffers, to adapt these algorithms to other codecs (particularly for time-compression) and to study the relevance of variance in the user perception.

The main conclusion of this paper is that the Frame Allocation algorithms are crucial to provide high-performance video delivery to fast clients under heavy load. Note also that this approach could be used on P2P/Ad-Hoc video streaming, where faster/better equipped peers/nodes could download frames to serve their slower neighbors.

References

1. Smith, B., Rowe, L.: Compressed domain processing of jpeg-encoded images. Real-Time Imaging 1(2), 3–17 (1996)
2. Bolot, J., Turletti, T.: Experience with control mechanisms for packet video. ACM Communication Reviews 28(1) (1998)
3. Floyd, S., Paxson, V.: Difficulties in simulating the internet. IEEE/ACM Transaction on Networking (Feburary 2001)
4. ITU-T: Video coding for low bitrate communication. Tech. rep., ITU-T Recommendation H.263, version 2 (January 1998)
5. JTC1, I.: Coding of audio-visual objects - part 2: Visual. Tech. rep., ISO/IEC 14496-2 (MPEG-4 Visual Version 1) (April 1999)
6. Zhang, L., Deering, S., Estrin, D., et al.: Rsvp: A new resource reservation protocol. IEEE Network 5, 8–18 (1993)
7. Paxson, V.: End-to-end internet packet dynamics. In: ACM SIGCOMM 1997, Cannes, France (April 1997)
8. Floyd, S., Fall, K.: Promoting the use of end-to-end congestion control in the internet. IEEE/ACM Trans. on Networking 7(4), 458–472 (1999)
9. Raghavan, S., Tripathi, S., Prabhakaran, B.: On qos parameters and services for multimedia applications. Tech. Rep. 3167, Dept. of Computer Science, Univ. of Maryland, MD (1994)
10. Varadarajan, S., Ngo, H., Srivastava, J.: Error spreading: A perception-driven approach to handling error in continuous media streaming. IEEE/ACM Trans. on Networking 10(1), 139–152 (2002)
11. Silberschatz, A., Galvin, P.B.: The Readers and Writers Problem. In: Operating System Concepts, 5th edn., pp. 173–175. Addison Wesley Longman, Inc., Amsterdam (1998)
12. Wiegand, T., Sullivan, G.J., Bjontegaard, G., Luthra, A.: Overview of the h.264/avc video coding standard. IEEE Trans. on Circuits and Systems for Video Technology 13(7) (July 2003)
13. Wenger, S.: H.264/avc over ip. IEEE Trans. on Circuits and Systems for Video Technology 13(7), 645–656 (2003)