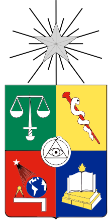


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IMPLEMENTACIÓN DE ALGORITMOS DE ASIGNACIÓN DE FRAMES EN UNA PLATAFORMA DE STREAMING DE VIDEO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

GONZALO MUÑOZ FERRER

PROFESOR GUÍA:
JOSÉ PIQUER GARDNER

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
RODRIGO PAREDES MORALEDA

SANTIAGO DE CHILE
SEPTIEMBRE 2013

RESUMEN DE LA MEMORIA PARA OPTAR AL TÍTULO DE: Ingeniero Civil en Computación
POR: Gonzalo Muñoz Ferrer
FECHA: 26/09/2013
PROFESOR GUÍA: José Piquer Gardner

IMPLEMENTACIÓN DE ALGORITMOS DE ASIGNACIÓN DE FRAMES EN UNA PLATAFORMA DE STREAMING DE VIDEO

Este trabajo continúa el estudio presentado en el paper *Frame Allocation Algorithms for Multi-threaded Network Cameras* (2010) de J. Piquer y J. Bustos-Jiménez. Este paper indaga sobre el efecto que tiene la selección de imágenes o *frames* de video en un servidor, para su transmisión a clientes a través de la red. Esta selección de frames ocurre al momento de asignar un frame para ser enviado a un cliente, y cuando se debe reemplazar uno de los frames guardados en la memoria del servidor por un nuevo frame obtenido desde la fuente de video. Las pruebas presentadas en el paper muestran que los algoritmos de asignación basados en contadores de referencias son los más efectivos, ya que permiten desocupar más rápidamente los frames que están en memoria para que puedan ser reemplazados por nuevos frames.

En el presente trabajo se implementaron los algoritmos de asignación y reemplazo de frames en la plataforma de streaming VLC, con el objetivo de usarlos para hacer streaming de video desde una fuente de video real hacia clientes reales, y comparar el desempeño de los distintos algoritmos con el rendimiento que actualmente ofrece VLC. Se realizaron pruebas para medir el número de *frames* por segundo (FPS) observado en los clientes al usar los algoritmos, y el uso de memoria y CPU en el servidor durante la transmisión de video.

Los resultados obtenidos muestran que, para los formatos de video que la implementación soporta, se obtiene una mejora importante en el rendimiento de VLC al usar los algoritmos de asignación y reemplazo adecuados, tanto en términos de FPS en los clientes como en uso de CPU en el servidor, utilizando en promedio una cantidad de memoria muy similar a la que usa VLC en condiciones normales. Se confirma además lo que la simulación realizada en el paper sugiere sobre el rendimiento de los algoritmos de asignación basados en los contadores de referencias, que en VLC ofrecen también el mejor desempeño.

Dedicado a mi hermosa familia: mi padre Álvaro, mi madre María Antonieta, mi hermano Álvaro y mi hermano Diego. Los amo.

Agradecimientos

Mi más profundo agradecimiento al profesor José Piquer, por su apoyo y excelente disposición durante la realización de este trabajo, y también por todo lo que aprendí de usted como estudiante.

A Javier Bustos y todo el equipo de NIC Chile Research Labs, por recibirme como lo hicieron en un ambiente muy grato y ameno de trabajo. Aprendí muchísimo y fue una experiencia muy valiosa.

A todos mis profesores y amigos del colegio, que han sido y seguirán siendo parte de mis más preciados recuerdos.

A mi padre, mi madre y mis dos hermanos, por su constante apoyo, preocupación y cariño.

Tabla de contenido

1 Introducción **1**
1.1 Problema a resolver . . . 2
1.2 Objetivos . . . 2
1.3 Motivación y relevancia del trabajo . . . 2
1.4 Descripción de contenidos . . . 3

2 Antecedentes **4**
2.1 Algoritmos de asignación y reemplazo . . . 4
2.2 Compresión y almacenamiento de video . . . 6
2.2.1 Codecs y formatos de video . . . 6
2.2.2 Contenedores . . . 7
2.3 Plataforma de streaming: VLC . . . 7
2.3.1 Pipeline de procesamiento . . . 8
2.3.2 Streaming HTTP en VLC . . . 8
2.4 Protocolos de transmisión de red . . . 10

3 Descripción de la solución **11**
3.1 Metodología . . . 11
3.2 Protocolo de transmisión . . . 11
3.3 Implementación . . . 12
3.3.1 Alternativa 1: Codificación única . . . 12
3.3.2 Alternativa 2: Codificación por cliente . . . 18

4 Pruebas y resultados **22**
4.1 Alternativa 1 . . . 23
4.1.1 Dos clientes rápidos . . . 23
4.1.2 Memoria virtual en *threads de cliente* . . . 31
4.1.3 Errores de decodificación . . . 32
4.1.4 Rendimiento de VLC estándar . . . 33
4.2 Alternativa 2 . . . 34
4.2.1 Dos clientes rápidos . . . 35
4.2.2 Errores de decodificación . . . 38

5 Conclusiones **39**

Bibliografía **41**

Índice de figuras

2.1 Algoritmo de asignación y reemplazo de frames 4
2.2 Pipeline de procesamiento de VLC 8
2.3 Streaming HTTP en VLC 9

3.1 Reemplazo de frames, Alternativa 1 de implementación 14
3.2 Asignación de frames, Alternativa 1 de implementación 16
3.3 Alternativa 1 de implementación 18
3.4 Reemplazo de frames, Alternativa 2 de implementación 19
3.5 Asignación de frames, Alternativa 2 de implementación 20
3.6 Alternativa 2 de implementación 21

4.1 Mediciones de FPS, 6, 8 y 10 buffers, Alternativa 1 24
4.2 Mediciones de FPS, 12, 15 y 20 buffers, Alternativa 1 25
4.3 Uso de CPU, 6 buffers, Alternativa 1 26
4.4 Uso de CPU, 8, 10 y 12 buffers, Alternativa 1 27
4.5 Uso de CPU, 15 y 20 buffers, Alternativa 1 28
4.6 Uso de memoria física, 6, 8 y 10 buffers, Alternativa 1 29
4.7 Uso de memoria física, 12, 15 y 20 buffers, Alternativa 1 30
4.8 Uso de memoria virtual, MAX-NF, Alternativa 1 31
4.9 Uso de memoria virtual al variar el stack de thread 32
4.10 FPS en el cliente más rápido, VLC estándar 34
4.11 Mediciones de FPS, MAX-NF, Alternativa 2 35
4.12 Uso de CPU, MAX-NF, Alternativa 2 36
4.13 Uso de memoria física, MAX-NF, Alternativa 2 37
4.14 Uso de memoria virtual, MAX-NF, Alternativa 2 37

Capítulo 1

Introducción

Las cámaras de red o “cámaras IP” reúnen la captura de video y la transmisión a través de Internet en un dispositivo único. Actualmente son ampliamente utilizadas para vigilancia y seguridad, facilitando el monitoreo a distancia sin introducir grandes costos en infraestructura. Uno de los desafíos que presentan estas cámaras es el servicio de múltiples clientes usando recursos limitados de memoria, y considerando las eventuales diferencias en la velocidad de la conexión de cada uno de estos clientes.

Este problema fue abordado en un *paper* de J. Piquer y J. Bustos-Jiménez [1]. La situación que se estudia es la de una cámara de red que atiende a múltiples clientes concurrentes, enviando *frames* de video capturados en vivo a cada uno. Cada vez que un frame es enviado a un cliente, se debe obtener desde la cámara el siguiente frame que se enviará.

Si la memoria de la cámara fuera suficiente para asignar un *buffer* a cada cliente conectado, la solución óptima consiste en disponer de un *thread de cámara* que obtiene frames desde la cámara y los escribe en cada buffer, y múltiples *threads de cliente* que envían el contenido de cada buffer a su respectivo cliente. De este modo, cada cliente recibe los frames tal como si fuera el único cliente conectado a la cámara, a la máxima velocidad que es capaz de mostrar.

Cuando hay menos buffers disponibles en memoria que clientes conectados, los clientes deben compartir un conjunto o *pool* de frames en memoria. Cada buffer en memoria guarda un frame de video, un registro de cuándo fue capturado y un contador de referencias que indica cuántos *threads de cliente* lo están usando. En este caso, durante el funcionamiento del servidor en general todos los buffers tendrán un frame guardado, y el *thread de cámara* debe decidir cuál frame que no esté siendo ocupado se sobrescribirá al obtener cada nuevo frame desde la cámara. Por otra parte cada *thread de cliente*, al no tener un buffer reservado en memoria, debe decidir qué buffer leer para enviar el siguiente frame a su cliente. Se debe entonces definir un *algoritmo de reemplazo de frames* (para escribir frames nuevos en memoria) y un *algoritmo de asignación de frames* (para asignar frames a los clientes). En el paper se propone una serie de algoritmos de asignación y reemplazo de frames, y se estudia su rendimiento.

1.1. Problema a resolver

Un aspecto importante que falta por estudiar es el efecto que tienen estos algoritmos al ser utilizados en una plataforma de *streaming* de video. Para abordar este tema, en el presente trabajo se implementan estos algoritmos en una plataforma de streaming de video, permitiendo su uso para hacer streaming desde una fuente real de video hacia dispositivos que efectivamente reciban y muestren un stream de video, considerando los cambios que sean necesarios para lograr esto, y se estudia su impacto.

Para elegir la plataforma de streaming apropiada se consideran alternativas *open-source* conocidas, dado que es necesario modificar el software para introducir la utilización de los algoritmos. Adicionalmente se reconocen características positivas del software open-source maduro, como su alta y creciente adopción por razones tanto estratégicas [2] como de costo [3], y su estabilidad [4].

1.2. Objetivos

El objetivo general de este trabajo es implementar los algoritmos de asignación y reemplazo de frames en una plataforma open-source de streaming de video. Los objetivos específicos son los siguientes:

- Medir el impacto de los algoritmos en la plataforma de streaming.
- Posibilitar el uso de los algoritmos para hacer streaming desde fuentes de video reales hacia clientes reales.
- Comprobar si efectivamente se logra una mejora al usar los algoritmos cuando hay recursos limitados y numerosos clientes concurrentes.

1.3. Motivación y relevancia del trabajo

El desarrollo de este trabajo involucra el estudio de distintas alternativas de software y la modificación de un software complejo ya existente, un desafío que muy probablemente se enfrenta en el ambiente laboral.

Los resultados que se obtienen de este trabajo proveen información importante para el desarrollo futuro de los temas estudiados en el paper, ya que se abordan aspectos que el estudio original no abordó. En el estudio presentado en el paper las pruebas se realizaron a través de simulaciones en el software Scilab y también se verificó que los resultados obtenidos en la simulación fueran consistentes con el comportamiento en una red real. Con esta premisa, el paper enfocó su trabajo en la administración de memoria bajo las condiciones anteriormente señaladas.

Por estas razones, dado que el algoritmo de asignación de frames fue validado en un simulador solamente, en este trabajo se presenta la primera implementación del algoritmo

en un sistema real, mostrando así su utilidad y una forma de distribución en un software de fuente abierta.

1.4. Descripción de contenidos

- **Antecedentes**: Información relevante sobre los conceptos y temas que se tratan en el documento, y las herramientas que se utilizan para el desarrollo del trabajo.
- **Descripción de la solución**: Detalle del funcionamiento y estructura de la solución que se implementa.
- **Pruebas y resultados**: Describe las pruebas que se realizan, y presenta los resultados que se obtienen en cada una de ellas.
- **Conclusiones**: Presenta las conclusiones que se obtienen a partir del trabajo, y un análisis de los objetivos, limitaciones y posibles trabajos futuros que se identifican.

Capítulo 2

Antecedentes

2.1. Algoritmos de asignación y reemplazo

El paper *Frame Allocation Algorithms for Multi-threaded Network Cameras* de J. Piquer y J. Bustos-Jiménez [1] propone un esquema de transmisión de video a múltiples clientes concurrentes. En el servidor se dispone de un conjunto o *pool* de *buffers* acotado, y sobre cada uno se escribe un frame de la secuencia de video que se obtiene a partir de la fuente. Cada buffer tiene un *timestamp* que marca el momento en que fue obtenido el frame que contiene, y un *contador de referencias* que indica el número de clientes que están leyendo ese buffer.

El servidor es *multi-threaded*: utiliza un *thread de cámara* para obtener frames desde de video y escribirlos en el pool, y por cada cliente conectado al servidor dispone de un *thread de cliente* que se encarga de seleccionar frames desde el pool para enviar al cliente correspondiente.

La Figura 2.1 ilustra la configuración descrita.

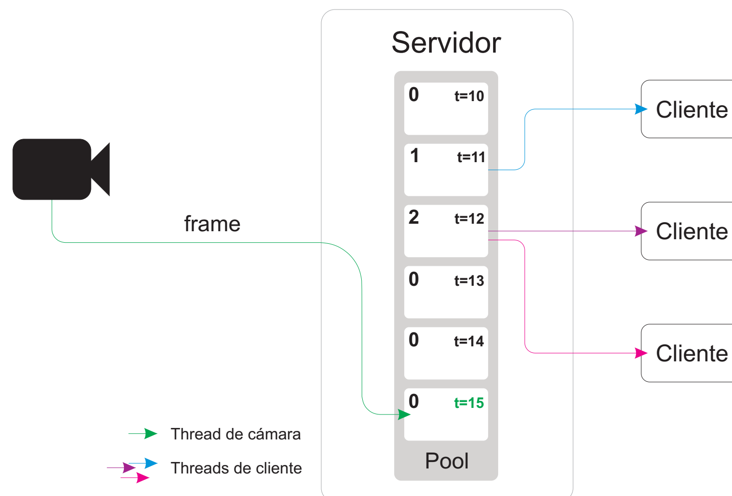


Figura 2.1: Algoritmo de asignación y reemplazo de frames

Dado que el número de buffers disponible es acotado, el *thread de cámara* utiliza un *algoritmo de reemplazo* de frames para decidir qué buffer sobreescribir cuando obtiene un nuevo frame, considerando que nunca debe escribir sobre un buffer que esté siendo utilizado por algún cliente (es decir, cuyo contador de referencias sea mayor a cero).

Por otra parte, los *threads de cliente* utilizan un *algoritmo de asignación* para obtener un frame desde el pool y enviarlo al cliente, considerando que al enviar un nuevo frame éste debe ser posterior al último frame enviado (es decir, debe tener un *timestamp* mayor).

En el paper se estudian los siguientes algoritmos de reemplazo de frames:

- **Oldest First** (**OF**): Se reemplaza el frame más antiguo.
- **Newest First** (**NF**): Se reemplaza el frame más reciente.
- **Any Frame** (**ANY**): Se reemplaza el primer frame disponible.

Los algoritmos de asignación de frames estudiados son los siguientes:

- **Oldest First** (**OF**): Envía al cliente el frame más antiguo en memoria.
- **Newest First** (**NF**): Envía al cliente el frame más reciente en memoria.
- **Maximum Reference Count** (**MAX**): Envía al cliente el frame con el mayor número de referencias. Si dos frames tienen el mismo número de referencias, se envía el más reciente.
- **Maximum Reference Count, Oldest First** (**MOF**): Envía al cliente el frame con el mayor número de referencias. Si dos frames tienen el mismo número de referencias, se envía el más antiguo.
- **Any Frame** (**ANY**): Envía al cliente un frame arbitrario.

Al combinar estos algoritmos y estudiar el efecto que tienen sobre el desempeño se concluye en el paper que, en términos de FPS (*frames per second*) en los clientes, el mejor algoritmo de asignación de frames es MAX, aunque genera mayor varianza en el número de frames omitidos[1]. La combinación NF/NF reduce significativamente el FPS de los clientes más rápidos, pero disminuye la varianza en el número de frames omitidos. Se concluye además que la elección del algoritmo de asignación de frames no afecta el FPS que recibe cada cliente, pero sí afecta qué frames recibe y por lo tanto la experiencia de cada usuario puede cambiar dependiendo del algoritmo utilizado. Una conclusión importante es que la elección de algoritmos de reemplazo y asignación es crucial para el servicio de clientes rápidos bajo alta demanda.

[1] Lo deseable es que la varianza en el número de frames omitidos sea baja. Esto significa que, por cada frame mostrado, la cantidad de frames omitidos se mantiene relativamente constante y la distancia entre cada frame mostrado es más uniforme, mejorando la experiencia del cliente.

2.2. Compresión y almacenamiento de video

2.2.1. Codecs y formatos de video

Un **codec** (*coder-decoder*, o *compressor-decompressor*) de video es un software capaz de codificar y decodificar una fuente de video usando un formato determinado. El objetivo generalmente es comprimir los datos, permitiendo que la transmisión y almacenamiento del video sea más eficiente. El formato de video define el esquema de compresión y almacenamiento de datos, mientras que el codec usa ese formato para codificar o decodificar.

Para lograr una buena compresión de datos la codificación normalmente es *lossy*, esto significa que en el proceso se descarta información del video original. Es posible, por ejemplo, reducir el detalle con que se codifica el color de la imagen, manteniendo el detalle de la luminosidad. Este proceso se llama *chroma subsampling*, y aprovecha la mayor sensibilidad de la visión humana a las diferencias en luminosidad que a las diferencias de color [12].

La calidad del video codificado resultante depende no solo de la cantidad de información que se descarta de la fuente de video, sino también de qué información se descarta y cómo se reconstruye el video al momento de su decodificación. En este sentido se identifican dos tipos de compresión, dependiendo de la información que se usa para la codificación: *intraframe* e *interframe* [13].

La compresión *intraframe* utiliza la información contenida en un único frame de video para codificar el frame resultante. *Chroma subsampling* es un ejemplo de compresión *intraframe*.

La compresión *interframe*, en cambio, aprovecha la redundancia de información que generalmente existe entre imágenes consecutivas en una secuencia de video, usando un frame como referencia para codificar otro frame de la secuencia (que puede ser anterior o posterior en el tiempo). El video codificado utilizando compresión *interframe* se compone de distintos tipos de frames de video:

- *I-frame*: Contiene una imagen completa.
- *P-frame*: Contiene una imagen parcial. Guarda solo la diferencia de imagen con respecto al frame anterior en la secuencia.
- *B-frame*: Contiene una imagen parcial. Usa un frame anterior como referencia (al igual que un P-frame) y un frame posterior.

Mientras más P-frames o B-frames haya en el video codificado, mayor es la compresión que se logra, pero esto depende de cuánta redundancia de información haya efectivamente en el video original.

Motion JPEG

Motion JPEG (M-JPEG) es un formato de video en que cada frame de video se codifica por separado usando el formato de compresión JPEG. Por esta razón, solo utiliza compresión *interframe*. Además de *chroma subsampling*, también se utiliza un método de compresión llamado *cuantificación*, que aprovecha la menor sensibilidad de la visión humana a cambios

de luminosidad de alta frecuencia. Al aplicar la cuantificación, las áreas de la imagen de mayor frecuencia (es decir, aquellas donde el detalle de la imagen es más fino) se codifican con menor precisión que las de menor frecuencia, reduciendo el tamaño de la imagen resultante sin que la pérdida de información sea demasiado notoria.

MPEG-1

El formato MPEG-1 utiliza las mismas técnicas de compresión *intraframe* de M-JPEG, y además utiliza compresión *interframe*. Para ello, se utilizan P-frames y B-frames que codifican solo la diferencia de imagen respecto a los frames de referencia, y además se utiliza una técnica denominada *estimación de movimiento* (EM). Esta técnica aprovecha la redundancia entre frames consecutivos que resulta del movimiento de la cámara o de los objetos en el video. Al usar EM, las pequeñas diferencias en la posición de los objetos entre dos frames se codifican utilizando un *vector* que indica, para una área determinada de la imagen, la dirección y magnitud del desplazamiento de esa área. La codificación de estos vectores requiere mucha menos información de la que se necesitaría para codificar un frame completo, permitiendo de esta forma reducir considerablemente el tamaño del video resultante.

2.2.2. Contenedores

Un video generalmente tiene dos componentes distintos: imagen y sonido. Estos componentes se unen, por ejemplo, en un archivo único o en una transmisión de red única usando un formato *contenedor*. Este formato define la forma en que distintos tipos de datos coexisten. En el caso particular de un video un contenedor puede unir, además de imagen y sonido, subtítulos, información de capítulos o distintos idiomas para el sonido, entre otros. El contenedor define además la forma en que se debe sincronizar los distintos tipos de datos al momento de su reproducción.

2.3. Plataforma de streaming: VLC

VLC es un software open-source multifuncional de reproducción, codificación y streaming de múltiples formatos de video y sonido. Es mantenido por la fundación VideoLAN, y su desarrollo comenzó en 1996 como un proyecto estudiantil en la École Centrale Paris [5][6].

El código de VLC está escrito en el lenguaje C, y tiene una arquitectura modular. Esto permite cargar distintos módulos de manera dinámica dependiendo de las necesidades, y extender el software para soportar nuevos formatos y protocolos de transmisión. El núcleo o *core* de VLC se encarga de instanciar los módulos requeridos para, por ejemplo, acceder a una fuente (como una cámara web), utilizar un codec de video determinado o guardar el video en un archivo. Se genera así un *pipeline* que procesa el contenido en etapas, cada una a cargo de un módulo específico [7].

Los módulos de VLC se categorizan según su *capacidad*, lo que permite a VLC determinar qué módulos son aptos para cada tarea [8].

2.3.1. Pipeline de procesamiento

Las etapas de procesamiento en VLC son las siguientes:

1. *Demux*: Separa el stream en sus componentes principales, video y sonido, y otros si los hubiera (como subtítulos o información de capítulos).
2. *Decode*: Decodifica cada componente del stream.
3. *Encode*: Codifica cada componente del stream.
4. *Mux*: Pone los componentes ya codificados del stream en un contenedor único.
5. *Output*: Envía el stream a su destino usando un módulo de salida.

Las etapas de decodificación y codificación se incluyen en el pipeline solo cuando es necesario cambiar el formato del stream. Un stream de video primero debe ser decodificado para obtener imágenes completas, dado que puede contener imágenes parciales (P-frames y B-frames), y a partir de la secuencia de imágenes completas codificarlo utilizando el nuevo formato. Este proceso en VLC está a cargo del módulo *transcode*, que a su vez instancia a los módulos de decodificación y codificación que se necesitan. El proceso se ilustra en la Figura 2.2.

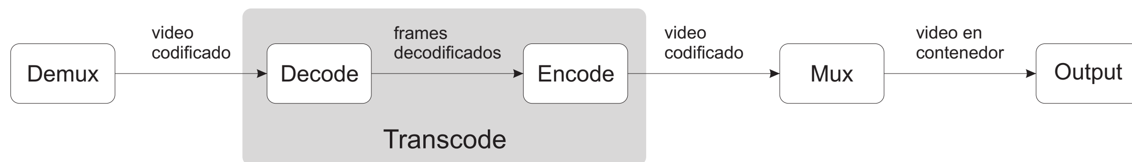


Figura 2.2: Pipeline de procesamiento de VLC

Por otro lado, si el formato de la fuente de video puede ser transmitido por el módulo de salida y si el usuario así lo prefiere, las etapas de decodificación y codificación pueden ser omitidas y el video llega a la etapa de *output* en su formato original, usando un contenedor adecuado.

2.3.2. Streaming HTTP en VLC

Los *módulos de salida* son los que normalmente se encuentran al final del pipeline de procesamiento y se encargan de llevar el contenido a su destino final. VLC permite hacer streaming a través de la red usando módulos de salida que utilizan distintos protocolos de transmisión, y lo hacen a través de *HTTPd* (*HTTP daemon*), el servidor web de VLC. HTTPd es parte del *core* o núcleo de VLC y provee las funcionalidades de bajo nivel para enviar y recibir datos y administrar las conexiones de cada cliente.

El módulo de salida HTTP implementa el streaming HTTP en VLC. Al usar este módulo, VLC puede recibir conexiones HTTP de múltiples clientes y hacer streaming de cualquiera de las fuentes que puede leer, tales como archivos de video locales, tarjetas de captura de video o cámaras web.

El módulo HTTP cumple dos funciones principales. La primera es configurar el servidor HTTPd e iniciar el *host thread* encargado de atender clientes. Su segunda función es copiar los datos recibidos desde el pipeline de VLC al buffer principal de HTTPd.

Al usar el módulo HTTP, HTTPd atiende a los clientes en un *modo de stream*. En este modo, la URL a la que se conecta un cliente se usa para asociarlo al stream que le corresponde, y se registra para esa URL una función *callback* que se encarga de pasar información desde el stream al cliente. Cuando un cliente pide una URL, HTTPd llama al callback correspondiente y envía al cliente los datos que el callback obtiene del stream, y desde ese momento el callback es llamado de manera sucesiva para seguir enviando datos al cliente sin que deba seguir haciendo peticiones. El proceso descrito se ilustra en la Figura 2.3.

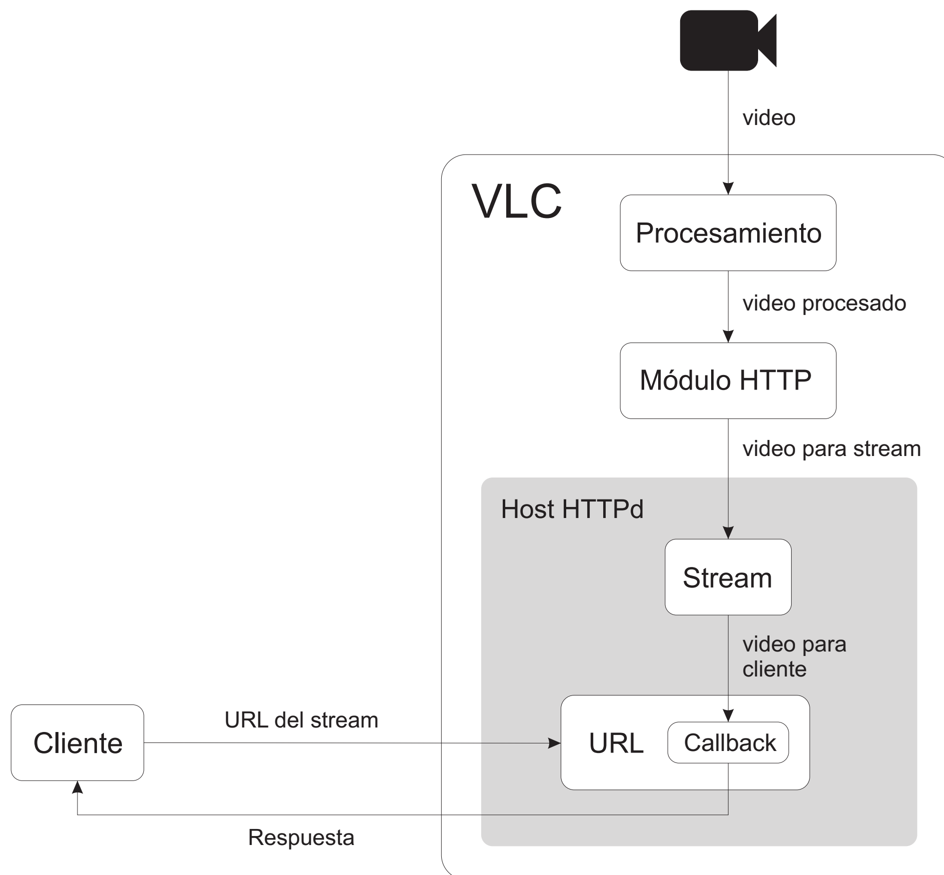


Figura 2.3: Streaming HTTP en VLC.

Cuando el callback obtiene datos desde el stream para enviar a un cliente, los copia a un buffer asignado exclusivamente a ese cliente, lo que permite que el módulo HTTP pueda seguir escribiendo sobre el buffer sin esperar a que se termine de enviar datos a cada cliente. Esto significa además que cada cliente necesita un buffer propio en el servidor VLC, aparte del buffer principal de HTTPd.

Se identifican los siguientes componentes del servidor HTTPd relevantes para la implementación de los algoritmos:

- **HostThread**: Thread encargado de atender clientes en *modo stream*. El módulo HTTP inicia este thread usando la función `httpd_HostThread` como *entry point*. Esta función

guarda una lista de clientes, cada uno con un estado propio. Los clientes son atendidos uno por uno de manera secuencial, y se usa la función callback para obtener desde el stream los datos que se envían a cada cliente.

- **ClientSend**: Función que se utiliza para enviar datos a un cliente. Esta función escribe sobre conexiones *no bloqueantes*, dado que es llamada por *HostThread* cada vez que se envían datos nuevos a un cliente y no debe bloquearse esperando que el cliente pueda recibirlos.
- **Stream**: Estructura que representa una fuente de video en transmisión. En esta estructura se guarda el buffer principal sobre el que se escriben los datos que deben ser transmitidos. Este buffer tiene un tamaño fijo de 5 Mo [2] y es circular: cuando la escritura sobre el buffer llega hasta su posición final, se comienza a sobreescribir desde el principio. Por cada cliente se guarda la posición absoluta (es decir, relativa al total de datos escritos en el buffer y no al tamaño del buffer) en que se encuentra leyendo datos, y se usa esa posición para determinar si un cliente es muy rápido (y debe esperar nuevos datos) o muy lento (y debe actualizarse su posición a la más reciente escrita en el buffer, descartando frames en el proceso).
- **StreamSend**: Función que utiliza el módulo HTTP para escribir datos en el buffer de un *stream*.

2.4. Protocolos de transmisión de red

Para transmitir video a través de una conexión de red es importante tener en cuenta el protocolo de transmisión utilizado, considerando especialmente qué características deseables de una conexión prioriza cada uno.

El protocolo de transmisión más utilizado es TCP (*Transmission Control Protocol*). TCP cuenta con mecanismos transparentes para las aplicaciones que permiten recuperarse en caso de haber datos dañados, perdidos, duplicados o desordenados en la transmisión. Provee facilidades para realizar control de flujo, permitiendo que un receptor maneje la cantidad de datos que recibe dependiendo de las características de su conexión [9]. TCP también hace control de congestión, monitoreando el estado de la conexión, detectando posibles sobrecargas de la red y tomando medidas cuando sea necesario para reducir la cantidad de datos que se envían y su frecuencia [10].

UDP (*User Datagram Protocol*) es otro protocolo de transmisión importante. A diferencia de TCP, UDP minimiza la intervención en la comunicación, y no garantiza que los datos enviados lleguen a su destino ni que éstos estén libres de errores, completos o en el orden correcto. Se evita de esta manera el costo adicional en tiempo de transmisión que deriva de corregir errores y controlar el flujo de la transmisión y la congestión de la red [11]. Por lo tanto cualquier corrección de errores, si es necesaria, es responsabilidad de las aplicaciones que usen UDP para enviar y recibir datos.

[2] 1 Mo (megaocteto) = 1,000,000 bytes

Capítulo 3

Descripción de la solución

3.1. Metodología

La metodología de trabajo es la siguiente:

1. Estudiar el funcionamiento de la transmisión de video a través de la red en VLC, identificando los componentes relevantes para la implementación de los algoritmos.
2. Definir el protocolo de transmisión de red más adecuado para la implementación.
3. Implementar los algoritmos de asignación y reemplazo de frames en VLC.
4. Definir y realizar pruebas que permitan examinar el funcionamiento de los algoritmos en VLC y comparar su rendimiento con el de VLC estándar. Las mediciones relevantes para estas pruebas son:
   - Tasa de frames por segundo (FPS) observada en los clientes.
   - Uso de CPU en el servidor.
   - Uso de memoria en el servidor.
   - Errores de decodificación en los clientes.

3.2. Protocolo de transmisión

Se considera que para implementar los algoritmos es más adecuado utilizar streaming HTTP en VLC, ya que permite aprovechar las garantías que ofrece el protocolo TCP. En particular, la garantía de confiabilidad de TCP se traduce en que un servidor, antes de continuar enviando datos nuevos, debe esperar a que su cliente cada cierto tiempo acuse recibo de los datos que ha recibido.[1] Esta lógica coincide con el comportamiento del servidor

[1] Técnicamente la aplicación solo espera hasta que tenga espacio en el buffer del *kernel* para escribir los datos que desea enviar, y TCP (en la capa de transporte) asume la responsabilidad de enviar efectivamente esos datos. Se libera espacio en el buffer a medida que se transmiten los datos, permitiendo así a la aplicación continuar escribiendo datos.

que se implementa, ya que solo debe enviar un frame nuevo a un cliente cuando el frame anterior ya ha sido enviado.

3.3. Implementación

Para implementar los algoritmos de asignación y reemplazo de frames en VLC, se identifican dos alternativas dependiendo del nivel del pipeline de procesamiento de VLC en que se insertan:

1. **Codificación única:** Los algoritmos se insertan en el módulo de salida HTTP y en el servidor HTTPd, al final del pipeline de procesamiento de VLC. En este escenario, los algoritmos actúan en una etapa posterior a la codificación, permitiendo que ésta se deba realizar solo una vez.
2. **Codificación por cliente:** Los algoritmos se insertan en la etapa de decodificación de VLC. Esto significa que los algoritmos administran las imágenes decodificadas, y por lo tanto cada imagen debe ser codificada por separado para cada cliente al que sea enviada.

A continuación se detalla la implementación de ambas alternativas.

3.3.1. Alternativa 1: Codificación única

Para esta alternativa de implementación se trabaja sobre el stream ya codificado. VLC maneja internamente los datos codificados en un formato de *bloques de datos* que proveen información adicional a los distintos módulos sobre su contenido. A través de *flags* presentes en cada bloque es posible saber si un bloque contiene un frame de video o si es un *header*, que es un tipo de bloque que contiene metadatos codificados en el stream. Se usa esta información de los bloques de datos para separar el stream codificado en frames, y así construir el `Pool` de frames que se utiliza en los algoritmos.

Se modifica el módulo HTTP para que al iniciar un stream configure el pool de buffers que se utiliza. Cada `Buffer` del pool se implementa como un `struct` que contiene una lista enlazada de bloques de datos que corresponden al mismo frame de video, y el `Pool` es a su vez un `struct` que contiene una lista enlazada de `Buffers`. Inicialmente, el `Pool` contiene una cantidad de `Buffers` vacíos que se definen al momento de iniciar el stream, y a medida que se procesa el video en el pipeline los `Buffers` se van llenando con frames. Cada `Buffer` guarda un *timestamp* que indica en qué momento se escribió el frame que contiene, y un contador de referencias que indica cuántos clientes se encuentran actualmente leyendo ese `Buffer`. Esta información es usada por los algoritmos para asignar o reemplazar `Buffers` según corresponda.

Para copiar frames al `Pool` se implementa una función **BufferedStreamSend** que se utiliza en vez de **StreamSend**. Esta función recibe bloques de datos desde el pipeline y, en vez de escribirlos sobre el buffer circular de HTTPd, lo hace sobre los `Buffers` del `Pool`.

Los bloques que corresponden a frames de video se identifican por un *flag* que indica el tipo de frame que contienen: I-frame, P-frame o B-frame. Si se detecta alguno de esos flags, se obtiene un `Buffer` del `Pool` usando el algoritmo de reemplazo de frames y se escribe ese bloque en el `Buffer`.

Los bloques que contienen headers también poseen un flag especial que los identifica. Estos bloques se escriben siempre seguidos de un frame de video en un mismo `Buffer` para manejar correctamente el formato M-JPEG, ya que en este formato el stream contiene un header por cada frame de video, y en el header se indica el tamaño en *bytes* del frame. Si se escribieran los headers y sus frames correspondientes en `Buffers` distintos, los clientes recibirían headers sin frames o frames sin headers, dependiendo del `Buffer` que el algoritmo de asignación seleccione.

BufferedStreamSend funciona de la siguiente forma:

```
BufferedStreamSend( stream, block ) {
    pool = getPool( stream );
    if( isFrameOrHeader(block) && !add_to_current ) {
        WriteToBuffer( block, pool );

        /* si block es un header, el bloque siguiente
           se agrega a new_buffer */
        add_to_current = isHeader( block );
    }
    else
        /* block se agrega a current_buffer,
           en vez de obtener un buffer nuevo */
        addBlock( pool->current_buffer, block );
}
```

Para escribir un frame en un `Buffer` nuevo, en **BufferedStreamSend** se usa la función **WriteToBuffer** que implementa los algoritmos de reemplazo de frames. Por ejemplo, si se usa el algoritmo de reemplazo **Oldest First** (OF), **WriteToBuffer** obtiene un `Buffer` desocupado (es decir, cuyo contador de referencias sea igual a cero) y cuyo *timestamp* sea menor al resto de los `Buffers` del `Pool`. El proceso se ilustra en la Figura 3.1.

El thread de VLC que lleva a cabo este proceso de escritura de frames corresponde al *thread de cámara* caracterizado en el paper. Esto significa que **WriteToBuffer** debe esperar cada vez que intenta escribir un frame en el `Pool` y no hay `Buffers` cuyo contador de referencias sea igual a cero.

WriteToBuffer funciona de la siguiente forma:

```
WriteToBuffer( block, pool ) {
    write_buffer = NULL;
    lock( pool );
    while( !(write_buffer = get_write_buffer(block,pool)) )
        wait( pool->ref_count );
    write_buffer->first_block = block;
    write_buffer->timestamp = time();
    pool->current_frame = write_buffer;
    signal( pool->new_timestamp );
    unlock( pool );
}
```

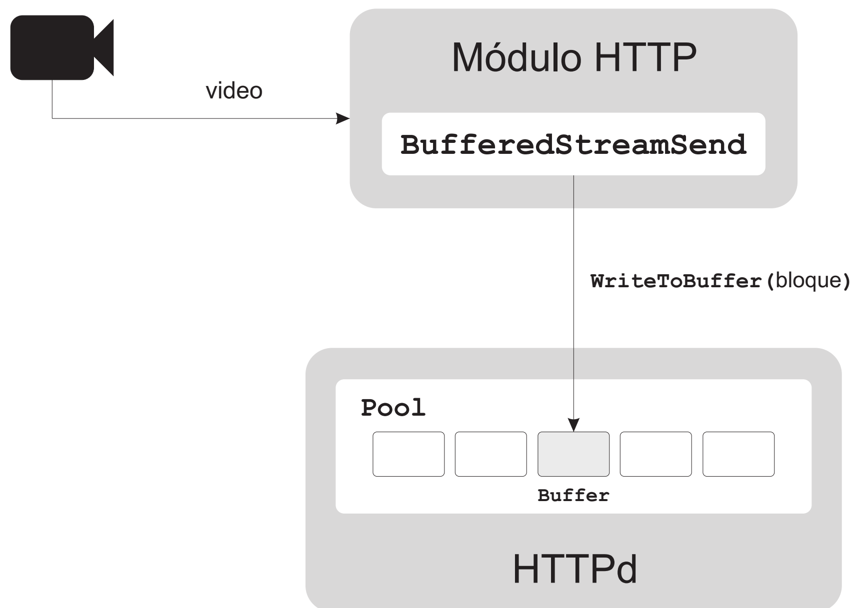


Figura 3.1: Reemplazo de frames, Alternativa 1 de implementación. El *thread de cámara* usa la función *WriteToBuffer* para obtener un *Buffer* del *Pool*, y escribe el bloque de datos en ese *Buffer*.

```
get_write_buffer( block, pool ) {
    buffer = pool->first;
    write_buffer = NULL;
    while( buffer ) {
        if( buffer->ref_count == 0 ) {
            if( !write_buffer ) {
                write_buffer = buffer;
                buffer = buffer->next;
                continue;
            }
            switch( pool->replacement_alg ) {
                case NF:
                    if( buffer->timestamp > write_buffer->timestamp )
                        write_buffer = buffer;
                    break;
                case OF:
                    if( buffer->timestamp < write_buffer->timestamp )
                        write_buffer = buffer;
                    break;
                case ANY:
                    break;
            }
        }
        buffer = buffer-next;
    }
    return write_buffer;
}
```

Aquí es necesario hacer una pequeña precisión. En el paper, el *thread de cámara* debe primero buscar un buffer desocupado en el pool de buffers, y después obtiene un frame desde la cámara. En esta implementación, en cambio, primero se obtiene un frame y luego se

busca un `Buffer` para escribirlo. Se hace así porque el módulo HTTP se encuentra al final del pipeline de procesamiento de VLC, y por lo tanto debe distribuir video ya procesado a sus clientes. Una consecuencia de esta implementación es que el *thread de cámara*, luego de esperar que haya un buffer desocupado, escribe sobre ese buffer un frame que estará atrasado en comparación con los frames que se están recibiendo desde la fuente de video. Este retraso se compensa descartando frames mientras el *thread de cámara* espera: luego de escribir ese frame antiguo, los frames siguientes que escribe el *thread de cámara* son los más recientes, ya que no se acumulan mientras espera.

Para atender clientes es necesario hacer dos modificaciones principales a VLC. La primera es en el funcionamiento del host HTTPd, que normalmente guarda una lista de clientes conectados y los atiende de manera secuencial. Se crea para esto una función **BufferedHostThread** que se utiliza para iniciar un thread que reemplaza al host HTTPd. A diferencia de éste último, **BufferedHostThread** solo se encarga de recibir conexiones de nuevos clientes, y por cada una crea un nuevo *thread de cliente* que atiende a ese cliente.

BufferedHostThread funciona de la siguiente forma:

```
BufferedHostThread( host ) {
    lock( host );
    while( host->running ) {
        remove_dead_clients( host );
        if( (connection = get_connection()) ) {
            client = ClientNew( connection, host->stream );
            add_client( host->clients, client );
            start_thread( ClientThread, client );
        }
    }
    remove_running_clients( host );
}
```

La segunda modificación se hace en el tipo de conexión que se establece entre el servidor y cada cliente, que en VLC normalmente es *no bloqueante*. Al usar este tipo de conexión, la función **send** que se utiliza para enviar datos retorna inmediatamente al ser llamada, es decir, la aplicación no espera a que los datos sean enviados para continuar trabajando. Esto es necesario en VLC porque el host debe atender al resto de los clientes y no puede esperar. Para el correcto funcionamiento de los algoritmos se debe utilizar conexiones *bloqueantes*, porque un *thread de cliente* solo se encarga de atender a un cliente y por lo tanto debe esperar a que los datos sean enviados antes de intentar enviar nuevos frames de video.

Cada cliente es atendido por un *thread de cliente* distinto, cuyo funcionamiento se implementa en la función **ClientThread**. Este thread se encarga de obtener frames desde el `Pool` usando la función **GetReadBuffer**, que implementa los algoritmos de asignación de frames, y permite obtener un frame para enviar al cliente correspondiente. Esta función además incrementa el contador de referencias del frame seleccionado. Por ejemplo, al utilizar el algoritmo de asignación **Newest First** (NF), **GetReadBuffer** busca el `Buffer` con el *timestamp* más reciente en el `Pool` y que sea también posterior al último frame enviado al cliente. El proceso descrito se ilustra en la Figura 3.2.

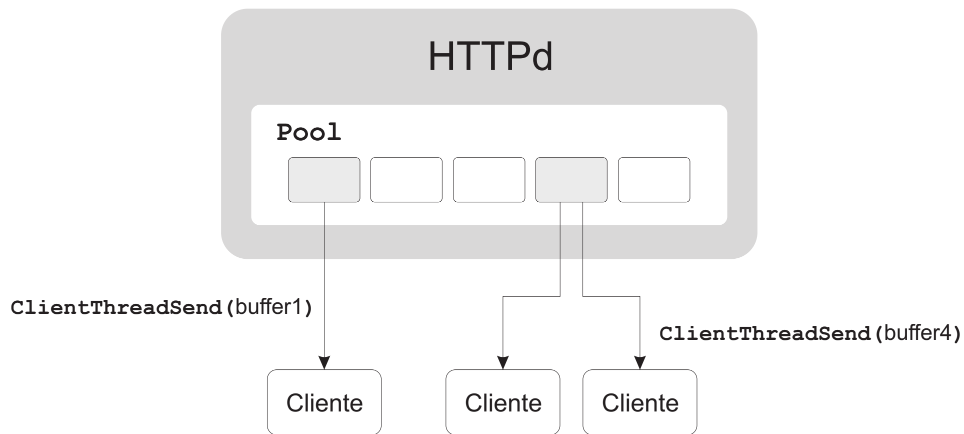


Figura 3.2: Asignación de frames, Alternativa 1 de implementación. Cada *thread de cliente* usa la función *GetReadBuffer* para obtener un *Buffer* del *Pool*, y usa la función *ClientThreadSend* para enviarlo al cliente que atiende.

La función **ClientThread** funciona de la siguiente forma:

```
ClientThread( client ) {
    stream = client->stream;
    pool = stream->pool;
    /* ts: timestamp */
    last_ts = -1;
    header_sent = false;
    while( is_alive(client) && is_running(stream) ) {
        if( header_sent ) {
            frame = GetReadBuffer( pool, last_ts );
            last_ts = frame->timestamp;
            ClientThreadSend( frame, client );
            ReleaseReadBuffer( frame, pool );
        }
        else {
            read_client_request( client );
            set_to_blocking_mode( client->connection );
            send_header( client, stream );
            header_sent = true;
        }
    }
}
```

GetReadBuffer funciona de la siguiente forma:

```
GetReadBuffer( pool, last_timestamp ) {
    read_buffer = NULL;
    lock( pool );
    while( !(read_buffer = get_read_buffer(pool,last_timestamp)) )
        wait( pool->new_timestamp );
    read_buffer->ref_count++;
    unlock( pool );
    return read_buffer;
}
```

```
get_read_buffer( pool, last_ts ) {
    read_buffer = NULL;
    buffer = pool->first;
    while( buffer ) {
        /* ts: timestamp */
        /* rc: reference count */
        if( buffer->ts <= last_ts )
            goto next;
        if( !read_buffer )
        {
            read_buffer = buffer;
            goto next;
        }
        switch( pool->allocation_alg ) {
            case OF:
                if( buffer->ts < read_buffer->ts )
                    read_buffer = buffer;
                break;
            case NF:
                if( buffer->ts > read_buffer->ts )
                    read_buffer = buffer;
                break;
            case MAX:
                if( buffer->rc > read_buffer->rc ||
                      buffer->rc == read_buffer->rc &&
                      buffer->ts > read_buffer->ts )
                    read_buffer = buffer;
                break;
            case MOF:
                if( buffer->rc > read_buffer->rc ||
                      buffer->rc == read_buffer->rc &&
                      buffer->ts < read_buffer->ts )
                    read_buffer = buffer;
                break;
            case ANY:
                break;
        }
        next:
            buffer = buffer->next;
    }
    return read_buffer;
}
```

La función **ReleaseReadBuffer** decrementa el contador de referencias del buffer que se libera, y despierta al *thread de cámara* cuando el contador de referencias llega a cero:

```
ReleaseReadBuffer( frame, pool ) {
    lock( pool );
    frame->ref_count--;
    if( frame->ref_count == 0 )
        signal( pool->ref_count );
    unlock( pool );
}
```

En la Figura 3.3 se ilustra el proceso completo de asignación y reemplazo de frames.

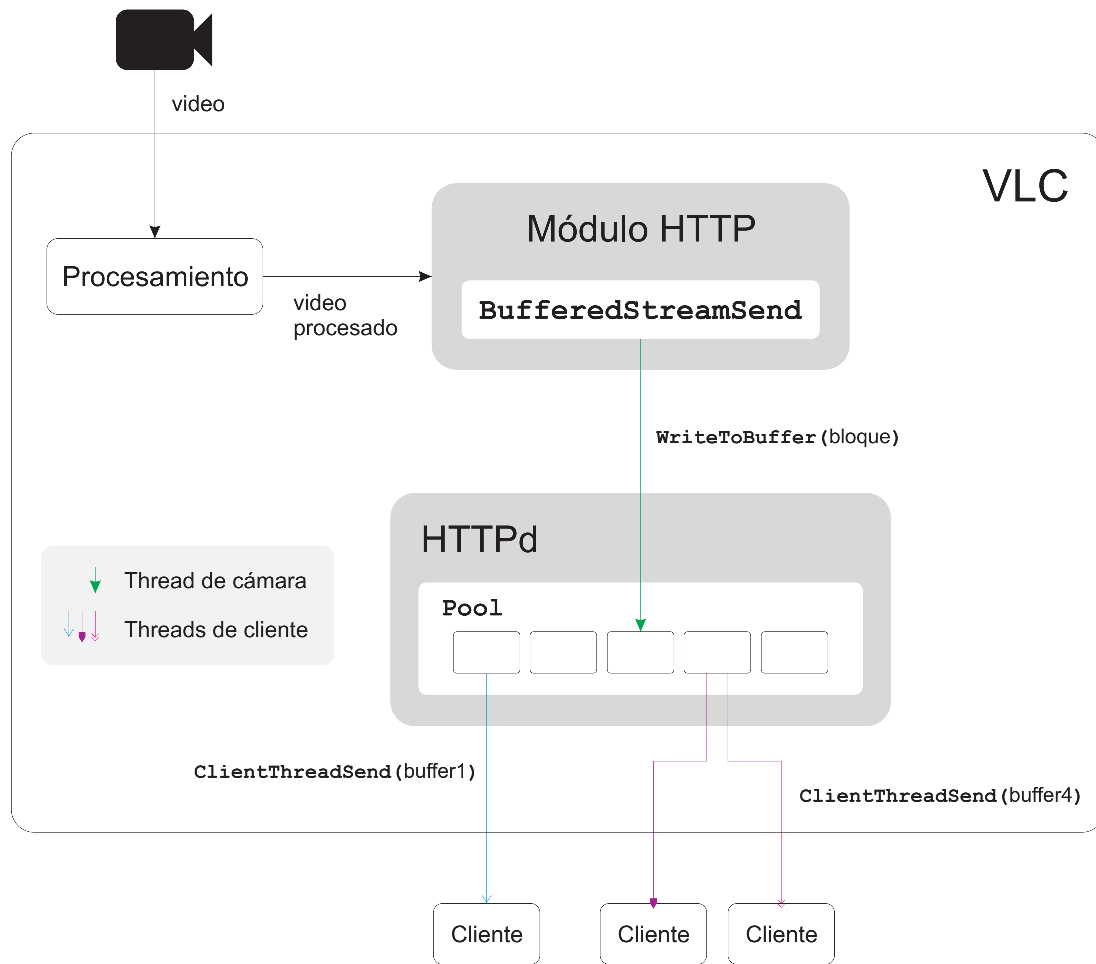


Figura 3.3: Alternativa 1 de implementación

Consideraciones

Se debe tener en cuenta que, si bien esta implementación funciona correctamente sin importar el formato del video, el resultado que se ve en los clientes es el esperado solo si el stream contiene únicamente I-frames. Esto sucede porque un cliente lento probablemente no recibirá la secuencia completa de frames y si se usan P-frames o B-frames, que requieren usar otros frames como referencia para su decodificación, el cliente no dispondrá de todos los frames de referencia que necesita para decodificar el stream correctamente. En consecuencia se utilizarán como referencia frames antiguos que el cliente sí recibió, generando "basura" en el video que el cliente mostrará.

3.3.2. Alternativa 2: Codificación por cliente

En esta alternativa de implementación los algoritmos se introducen en la etapa de decodificación de VLC. A diferencia de la Alternativa 1, en que el stream ya codificado se separa

en frames, en este caso se trabaja sobre el stream a medida que se decodifica, y por esta razón se tiene acceso a la secuencia de imágenes que resultan de este proceso. El objetivo es codificar cada imagen teniendo en cuenta a qué cliente es enviada, permitiendo que se usen como referencia solo frames que ya han sido enviados a ese cliente. Se evita de esta forma el problema que surge al usar al Alternativa 1 con P-frames o B-frames, ya que un cliente lento siempre recibe frames que puede decodificar usando los que ya ha recibido.

Los cambios se insertan específicamente en el módulo *transcode* de VLC, entre la decodificación y codificación del video. La estructura del `Pool` de `Buffers` es similar a la descrita en la Sección 3.3.1. La diferencia está en el contenido de cada `Buffer`, que en vez de ser una secuencia de bloques de datos ahora es una imagen completa, obtenida al decodificar el stream, y que todavía no ha sido procesada para su transmisión.

La función **WriteToBuffer** se usa ahora directamente desde el módulo *transcode* pero funciona de la misma forma que en la Alternativa 1, escribiendo imágenes en `Buffers` obtenidos desde el `Pool` usando un algoritmo de reemplazo de frames. La fuente de video no se procesa completamente antes de llegar al módulo HTTP, y solo se hace el proceso de *demux* y decodificación. Se ilustra este proceso en la Figura 3.4.

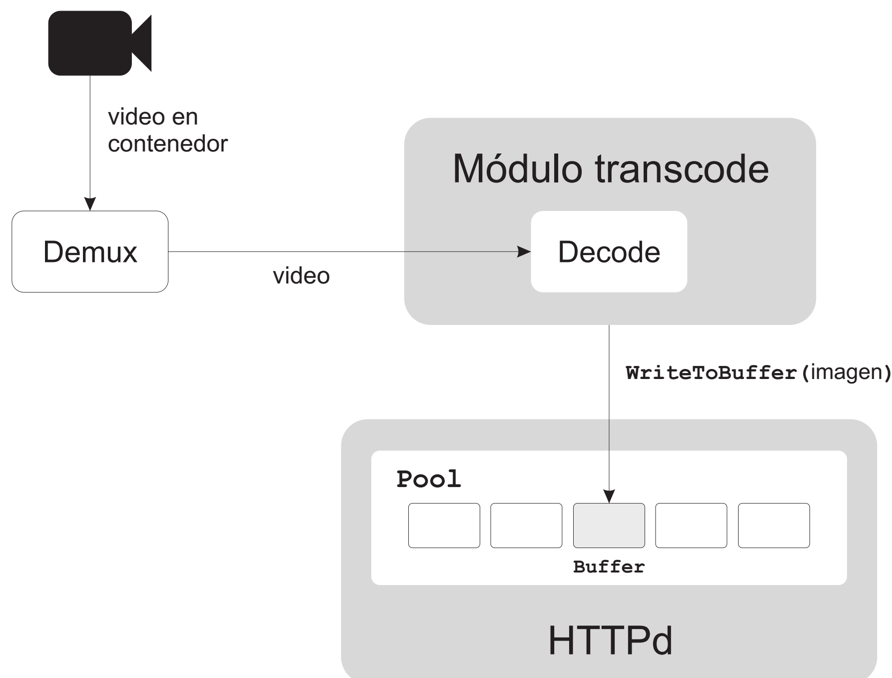


Figura 3.4: Reemplazo de frames, Alternativa 2 de implementación. El *thread de cámara* usa la función *WriteToBuffer* para obtener un *Buffer* del *Pool*, y escribe una imagen en ese *Buffer*.

La atención de clientes requiere más trabajo. Por una parte el host HTTPd se modifica para que se encargue solo de recibir conexiones e iniciar *threads de cliente*, al igual que en la Alternativa 1, pero la función que cumple cada *thread de cliente* es más compleja ya que cada frame obtenido con la función **GetReadBuffer** se debe codificar para enviarlo al cliente.

Para hacer la codificación, cada *thread de cliente* guarda un *contexto de codificación*, que es un `struct` proporcionado por la librería *libavcodec* que usa VLC para decodificar y codificar video. Este contexto guarda toda la información que necesita *libavcodec* para hacer

la codificación (parámetros, buffers internos, etc.), incluyendo una referencia al último frame que fue codificado.

Normalmente, dado que el stream se codifica solo una vez, durante la ejecución de VLC solo existe un contexto. En esta implementación cada imagen se codifica usando el contexto del cliente que la recibe, logrando que cada P-frame o B-frame producido contenga referencias solo a frames que ese cliente haya recibido anteriormente. Por esta razón se necesita crear un contexto por cada cliente, y se hace una codificación del stream por cada cliente conectado.

Después de codificar se hace el proceso de *muxing*, poniendo el frame en un contenedor apropiado para el formato de video utilizado, y finalmente se envía el frame al cliente. Se ilustra el proceso descrito en la Figura 3.5.

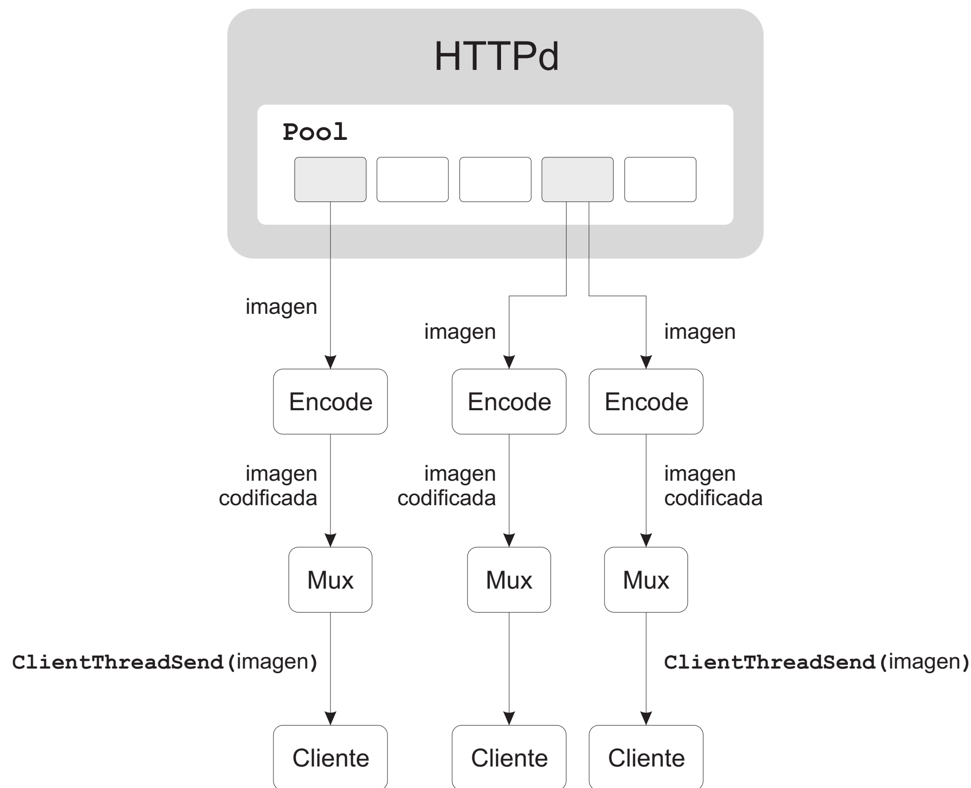


Figura 3.5: Asignación de frames, Alternativa 2 de implementación. Cada *thread de cliente* usa la función *GetReadBuffer* para obtener un *Buffer* del *Pool*. Luego hace la codificación (*encode*) y *muxing*, obteniendo un frame que envía al cliente usando la función *ClientThreadSend*.

Se ilustra el proceso completo de asignación y reemplazo de frames en la Figura 3.6.

Consideraciones

En la Alternativa 1 los algoritmos trabajan sobre un stream codificado, y por lo tanto solo es necesario codificar el stream una sola vez. En este caso, en cambio, cada cliente recibe una codificación del stream distinta que depende de su velocidad y de la cantidad de clientes. La ventaja de esta implementación es que permite utilizar los algoritmos de asignación y reemplazo de frames con formatos de video que utilicen compresión *intraframe*.

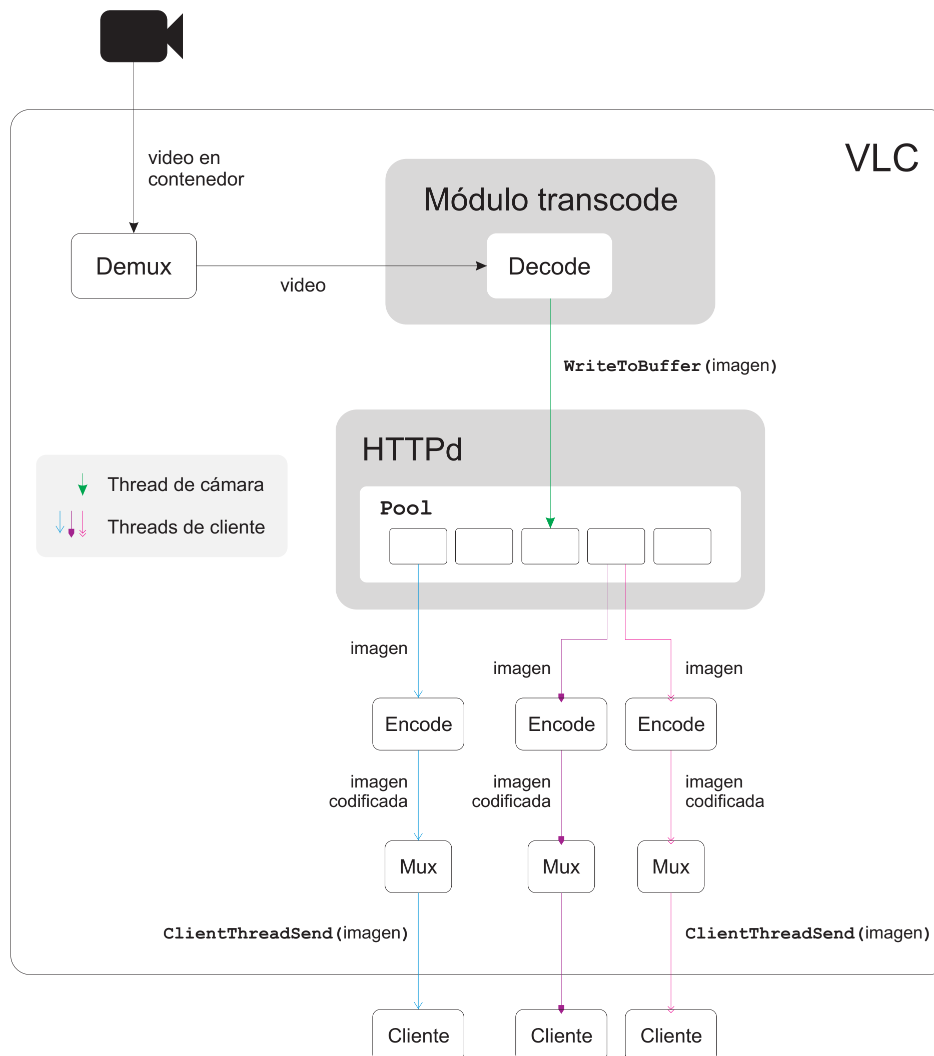


Figura 3.6: Alternativa 2 de implementación

En particular funciona correctamente cuando un stream contiene P-frames y B-frames, de esta forma reduciendo el ancho de banda de red necesario para la transmisión de datos a cada cliente ya que la mayor parte del tiempo no se envían imágenes completas.

La principal desventaja de hacer una codificación por cliente es que es un proceso muy intensivo en recursos. El uso de memoria aumenta considerablemente cada vez que el servidor VLC recibe una conexión, ya que debe crear un nuevo *contexto de codificación* para cada cliente y mantenerlo en memoria mientras ese cliente esté conectado. El uso de CPU crece también, porque el proceso de enviar datos a un cliente ya no solo involucra la transmisión de datos, sino también su codificación.

Capítulo 4

Pruebas y resultados

Se realizaron pruebas para medir el efecto que tiene el uso de los algoritmos de asignación y reemplazo de frames en VLC. Las pruebas se ejecutaron en un computador Dell Inspiron N4020, con procesador Pentium T4500 de 2.3 GHz Dual-Core y 2 GB de memoria RAM. Se usó el sistema operativo Ubuntu 11.10.

Se mide el uso de memoria y tiempo de CPU en el servidor VLC usando la herramienta **top**. Esta herramienta provee dos mediciones de memoria distintas [14], todas se obtienen a intervalos de un segundo y se calcula el promedio de cada métrica una vez terminada la ejecución de la prueba:

- Memoria virtual (VIRT): Memoria virtual que usa un proceso. Incluye el código, datos, librerías compartidas, y memoria que ha sido reservada pero no utilizada.
- Memoria residente (RES): Memoria física que usa un proceso.

Para hacer mediciones de *frames por segundo* (FPS) que recibe un cliente, se usan clientes VLC que pueden simular una conexión lenta, y se corre el servidor y los clientes en la misma máquina. Paras las pruebas con las Alternativa 1 se implementa la medición de FPS tanto desde los clientes como desde el servidor, mientras que en la Alternativa 2 se utilizan solo las mediciones obtenidas desde los clientes. En ambos casos, el FPS de cada cliente se mide contando la cantidad de veces que la ventana de video del cliente VLC se refresca para mostrar una nueva imagen, y se divide por el tiempo total en segundos transcurrido entre la primera y la última imagen mostrada. Para medir FPS desde el servidor al usar algoritmos se cuenta, por cada cliente, la cantidad de frames enviados, y se divide por el tiempo transcurrido entre el primer frame enviado y el último. Tanto el servidor como los clientes son ejecutados en la misma máquina.

La simulación de clientes lentos se hace definiendo una velocidad de conexión para cada cliente, en KiB[1]/segundo, y se usa esa velocidad para insertar un *delay* en la función **ReadData** proporcionada por VLC para leer datos desde una conexión de red. Si un cliente recibe X KiB desde el servidor y su velocidad es V KiB/s, el *delay* asociado es de $^{X}/_{V}$ segundos.

[1] 1 KiB (kibibyte) = 1024 bytes.

4.1. Alternativa 1

Para la realización de las pruebas con la Alternativa 1 de implementación se utilizó un video de resolución 854x480[2] y 29.97 FPS, codificado con el formato H.264. El video se convierte al formato M-JPEG, que utiliza únicamente compresión *intraframe* y por lo tanto produce un stream que contiene solo I-frames.

4.1.1. Dos clientes rápidos

En esta prueba se conectan múltiples clientes lentos y se mide el FPS en dos clientes rápidos, que pueden recibir video a la máxima velocidad posible. Los clientes lentos tienen velocidades distintas en el rango 149-151 KiB/s que se asignan aleatoriamente, recibiendo video en el rango 0.5-1.5 FPS, asemejando las condiciones estudiadas en el paper.

En cada prueba individual se conecta una cantidad definida de clientes lentos, que va desde 0 hasta 260. Por cada una se ejecuta el servidor y se conectan clientes cada 0.5 segundos. Esto se hace así porque cada cliente es un proceso pesado VLC, y por lo tanto ejecutar todos los clientes uno después de otro sin esperar produce saturación en la máquina de pruebas. Se hacen las mediciones de FPS, uso de memoria y uso de CPU durante 5 minutos desde el momento en que se conecta el último cliente. Solo los clientes rápidos muestran su ventana de video, lo que es necesario para obtener una medición de FPS en el cliente.

Se muestran los resultados obtenidos utilizando solo el algoritmo de reemplazo NF ya que en las pruebas realizadas se ve que, tal como se muestra en el paper, la elección de este algoritmo no influye en el FPS recibido por los clientes.

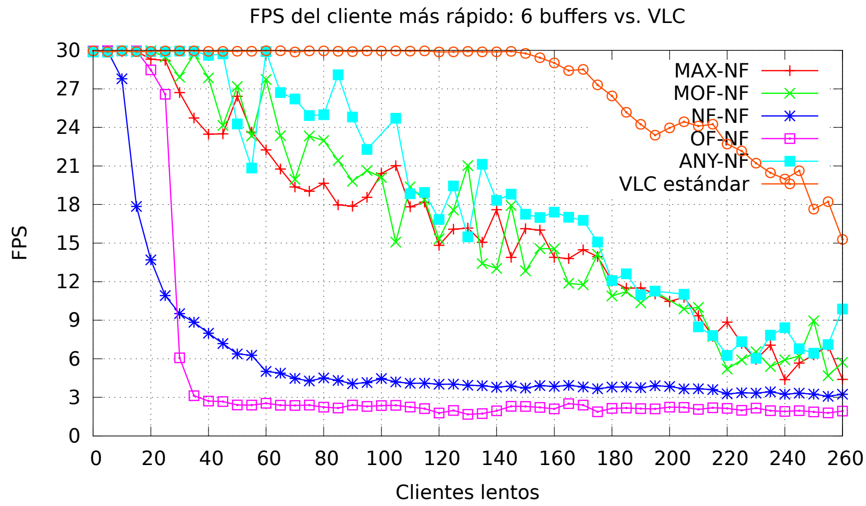
FPS

En las Figuras 4.1a y 4.1b se muestran las mediciones de FPS obtenidas en el cliente más rápido usando 6 y 8 buffers respectivamente. Se observa que al usar VLC estándar se obtiene siempre un FPS promedio mayor, sin importar el algoritmo utilizado.

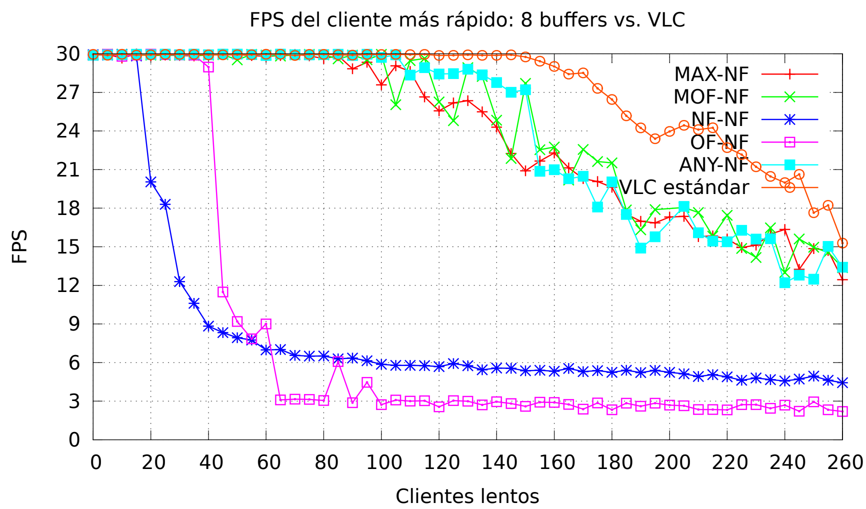
En la Figura 4.1c se muestran las mediciones de FPS obtenidas al usar 10 buffers. En este caso el comportamiento de VLC estándar es similar al de los algoritmos MAX y MOF (la diferencia se mantiene cercana a los 3 FPS), y mejor que el de los algoritmos NF y OF.

En las Figuras 4.2a, 4.2b y 4.2c se muestran las mediciones de FPS al usar 12, 15 y 20 buffers respectivamente. En las tres pruebas se observa que los algoritmos MAX, MOF y ANY se comportan mejor que VLC estándar. MOF supera a MAX, aunque la diferencia es menor (normalmente cercana a 2 FPS).

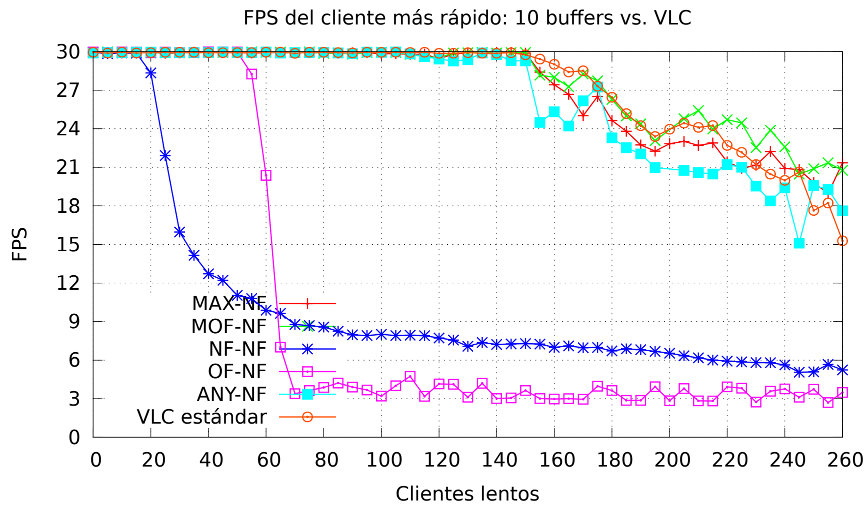
[2]La resolución 854x480 se denomina *Full Wide Video Graphics Array* (FWVGA) y corresponde a la resolución estándar de video DVD *widescreen*.



(a) 6 buffers

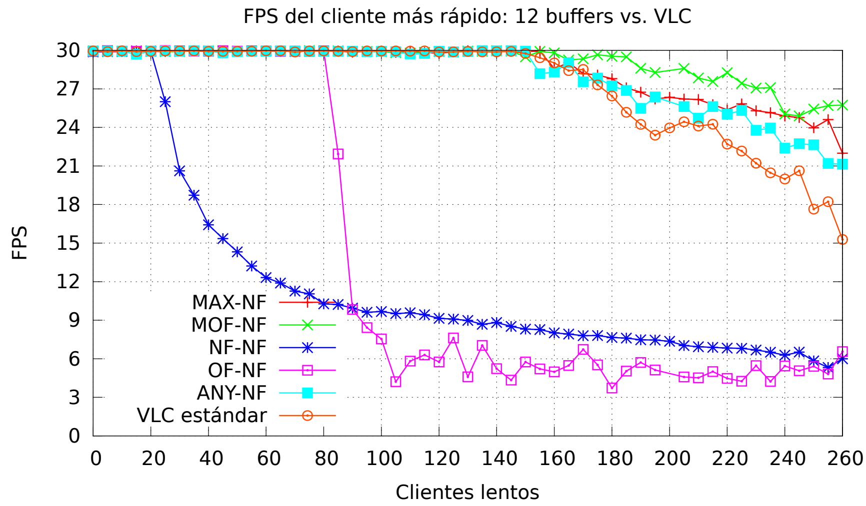


(b) 8 buffers

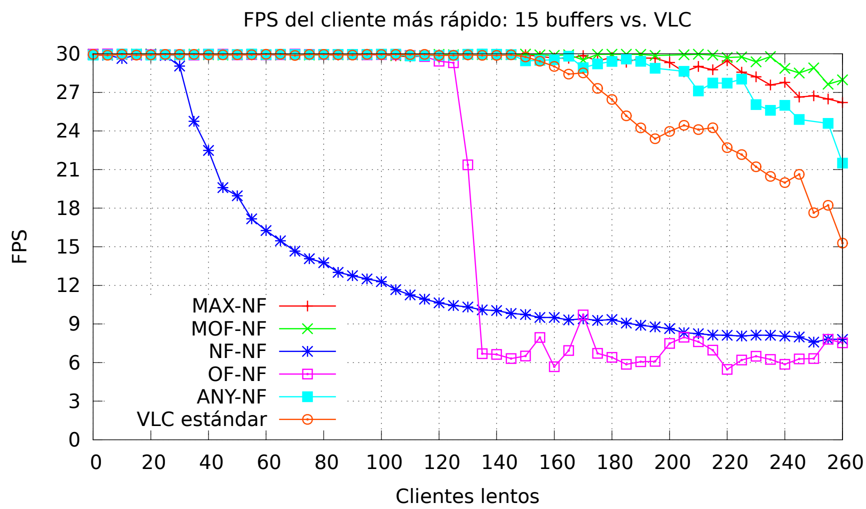


(c) 10 buffers

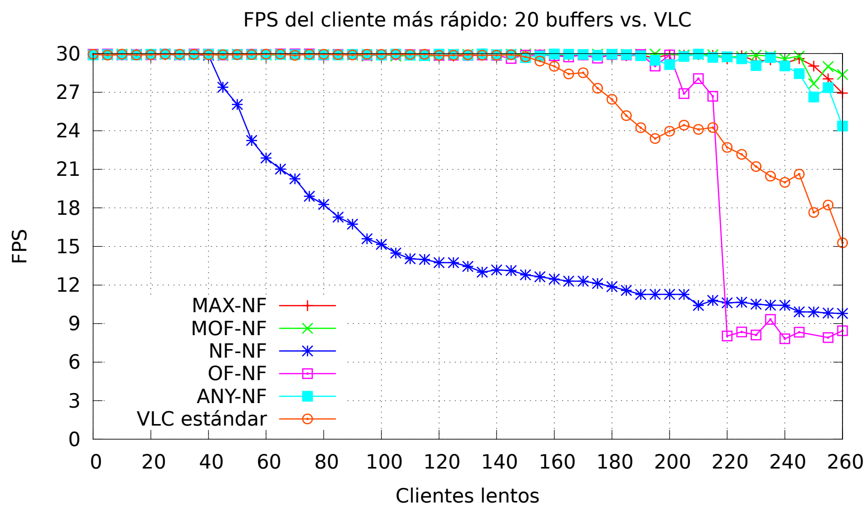
Figura 4.1: Mediciones de FPS con 2 clientes rápidos, usando 6, 8 y 10 buffers.



(a) 12 buffers



(b) 15 buffers



(c) 20 buffers

Figura 4.2: Mediciones de FPS con 2 clientes rápidos, usando 12, 15 y 20 buffers.

En todas las pruebas anteriores se observan tres diferencias con respecto a los resultados presentados en el paper. La primera es que no se observa la diferencia marcada que se mostraba en el paper entre el algoritmo MAX y el algoritmo ANY. Con 8 buffers, por ejemplo, la diferencia entre MAX y ANY no es nunca mayor a 6 FPS, mientras que en el paper el algoritmo MAX superaba por cerca de 15 FPS a ANY a partir de los 30 clientes. La segunda diferencia está en el comportamiento del algoritmo OF, que en el paper presentaba una disminución gradual del FPS promedio conforme se agregaban clientes lentos. En estas pruebas se observa una disminución mucho más brusca, y un intervalo acotado en que OF supera a NF que no se ve en las pruebas del paper. Finalmente se observa que al usar 6 y 8 buffers MAX no supera a MOF de manera consistente y marcada como en el paper.

Uso de CPU

En las Figuras 4.3 y 4.5 se muestran las mediciones de uso de CPU. Se observa que al usar algoritmos se usa siempre menos tiempo de CPU que al no usarlos, sin superar el 65 %, incluso en los casos en que se obtienen FPS mayores en el cliente más rápido (12, 15 y 20 buffers). Se ve además que el uso de CPU disminuye a medida que se agregan clientes y disminuye el FPS promedio. Esto sucede porque en esos casos, cuando hay muchos clientes conectados, es más probable que todos los buffers estén siendo leídos por algún cliente. Cuando esto pasa el *thread de cámara* del servidor debe bloquearse esperando que algún buffer se libere, y por lo tanto usa menos tiempo de CPU mientras espera.

Por estas razones la tendencia de los gráficos de uso de CPU está ligada a la de los gráficos de FPS, observándose que los tramos en que el FPS de los clientes rápidos es mayor coincide con aquellos en que el uso de CPU es más intenso.

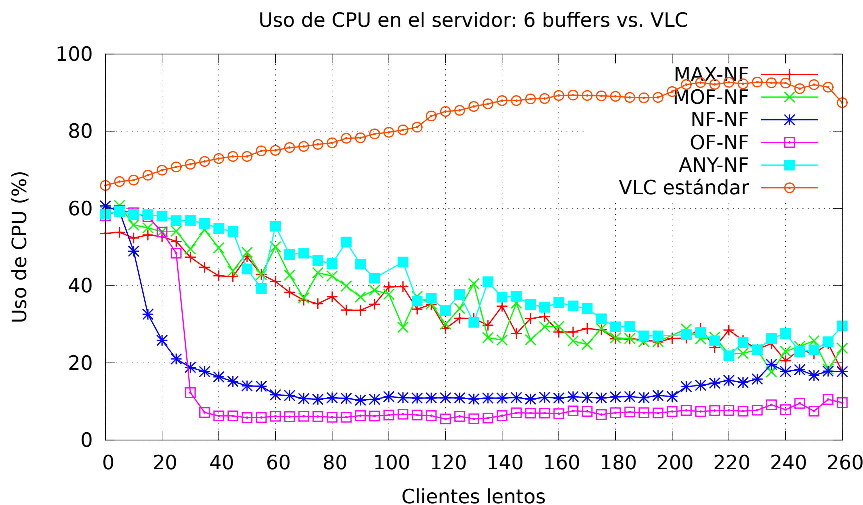
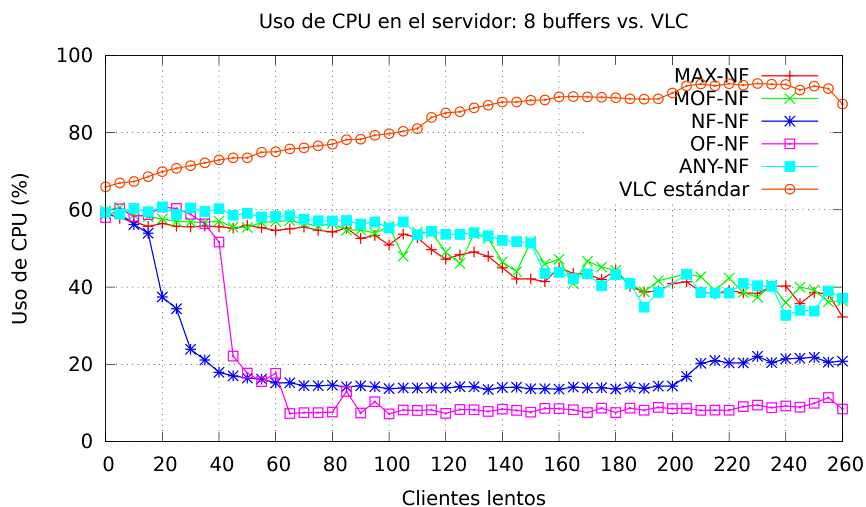
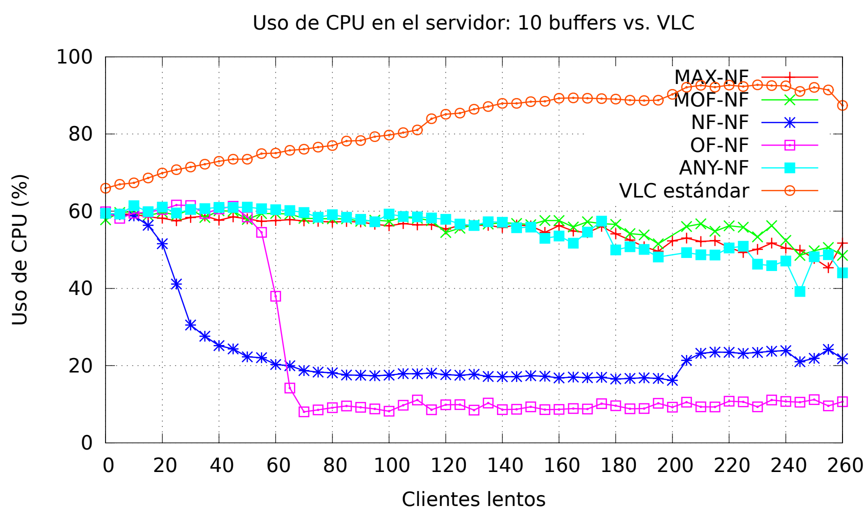


Figura 4.3: Mediciones de uso de CPU con 2 clientes rápidos, usando 6 buffers.





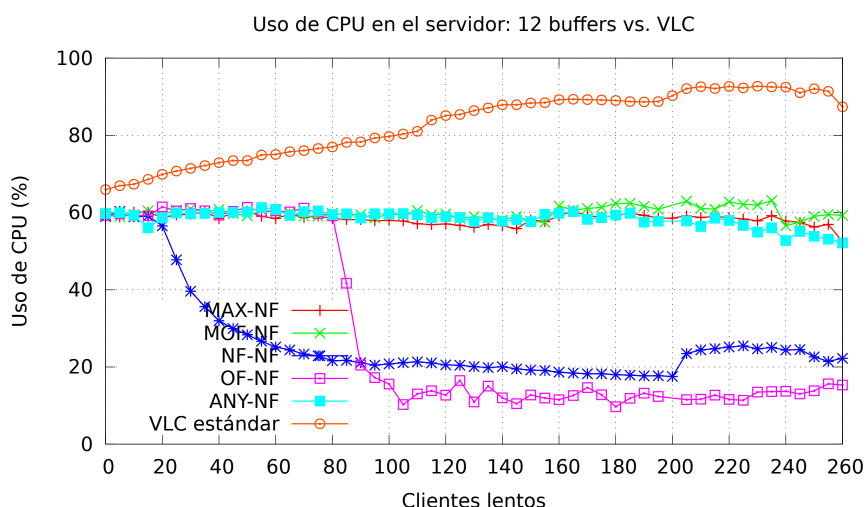


Figura 4.4: Mediciones de uso de CPU con 2 clientes rápidos, usando 8, 10 y 12 buffers.

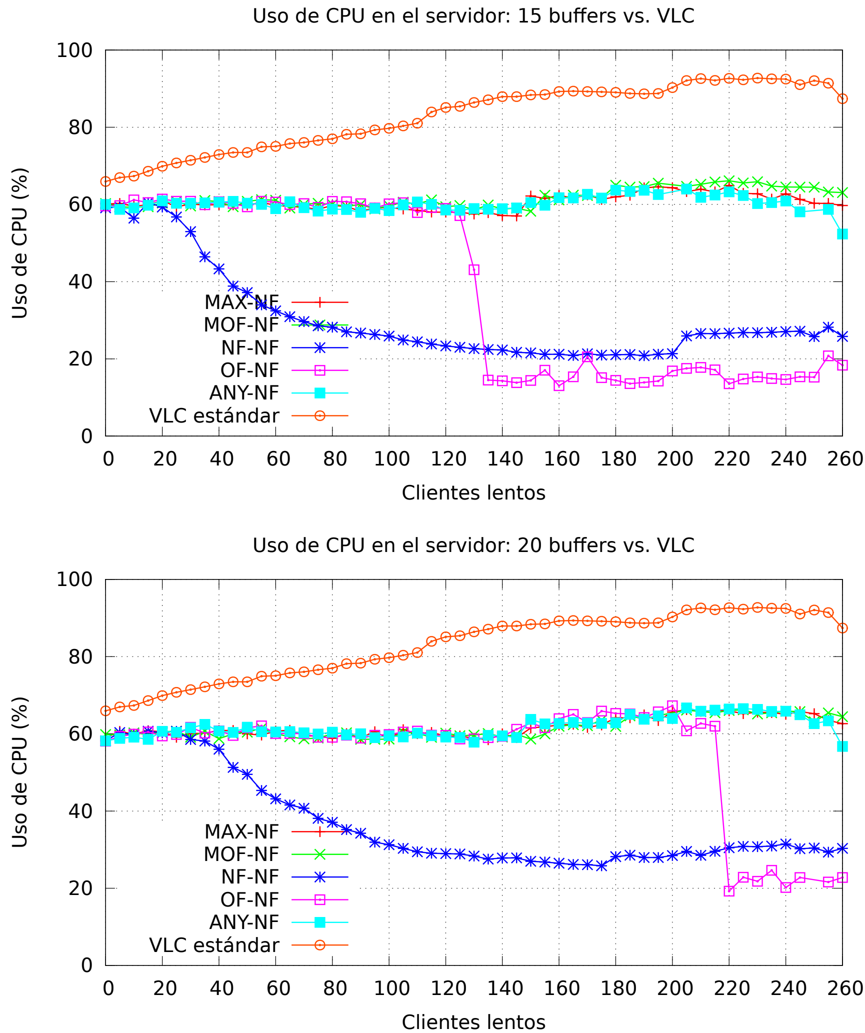
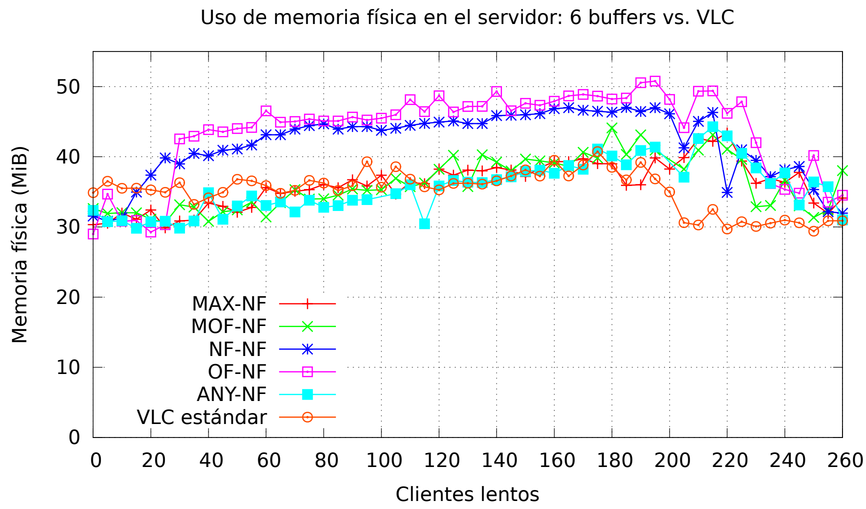
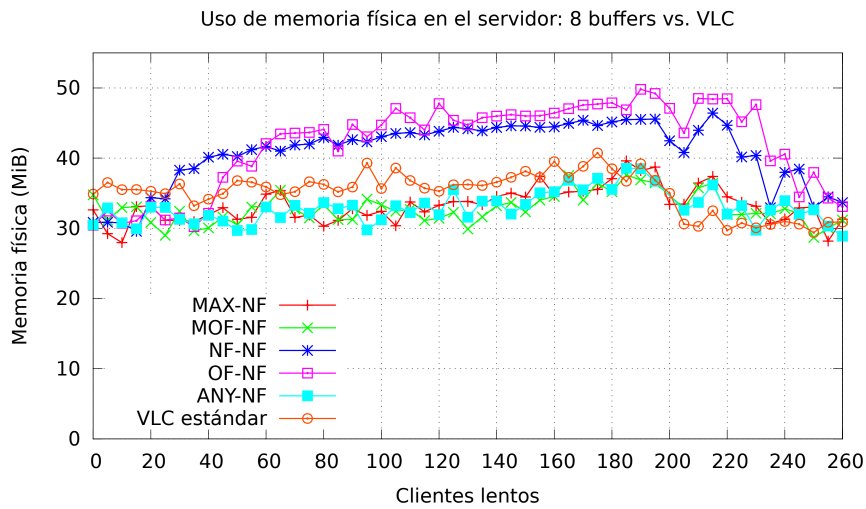


Figura 4.5: Mediciones de uso de CPU con 2 clientes rápidos, usando 15 y 20 buffers.

Uso de memoria física

En las Figuras 4.6 y 4.7 se muestra el uso de memoria física. Se observa que, al usar algoritmos, el uso de memoria disminuye a medida que se obtienen mejores FPS. Al usar 8 buffers, por ejemplo, el uso de memoria de los algoritmos MAX, MOF y ANY es muy similar, y menor al de los algoritmos NF y OF. Entre estos dos últimos, el que menos memoria usa es OF, excepto en el tramo en que el FPS que se obtiene con NF es mayor. Este mismo comportamiento se observa en el resto de las pruebas. Por otra parte se observa que el uso de memoria disminuye a medida que se agregan buffers. Este comportamiento, si bien parece contraintuitivo (ya que más buffers requieren reservar más memoria), es el resultado de la administración de memoria de todos los procesos que realiza el sistema operativo, y una de sus principales consecuencias es que la memoria que se reserva durante la ejecución de un programa no coincide con la memoria que efectivamente es utilizada por ese programa.





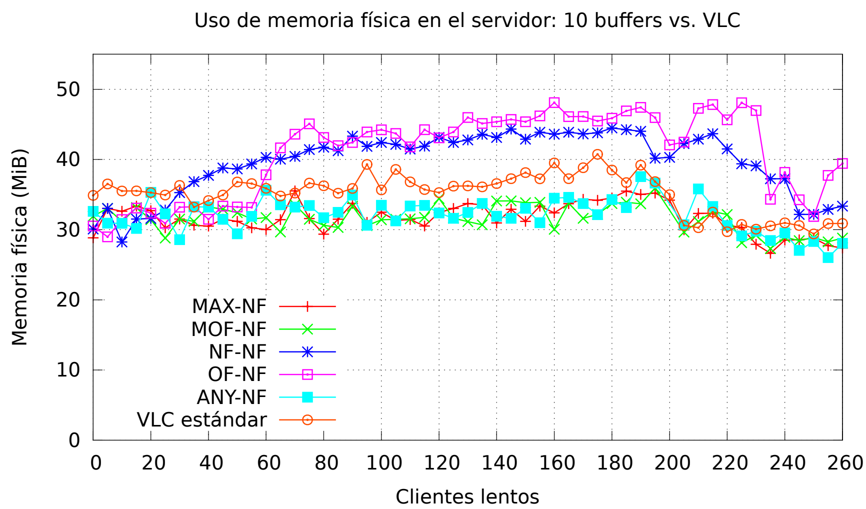
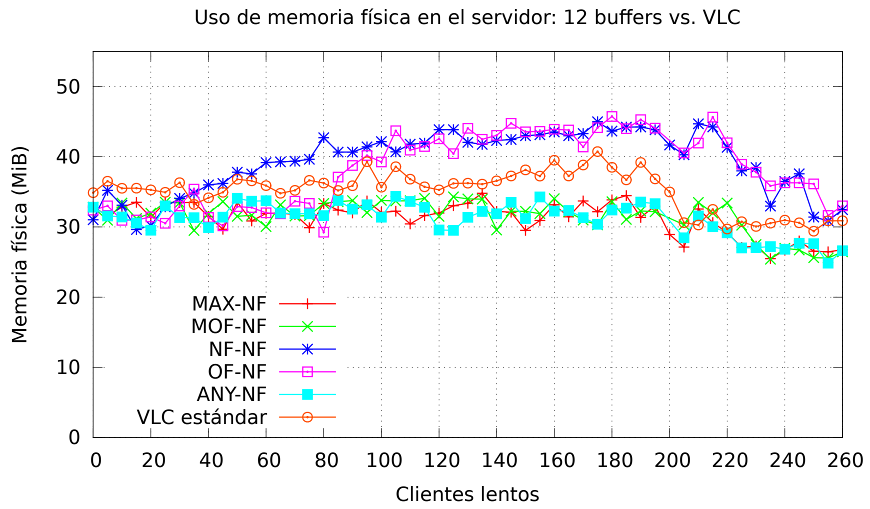
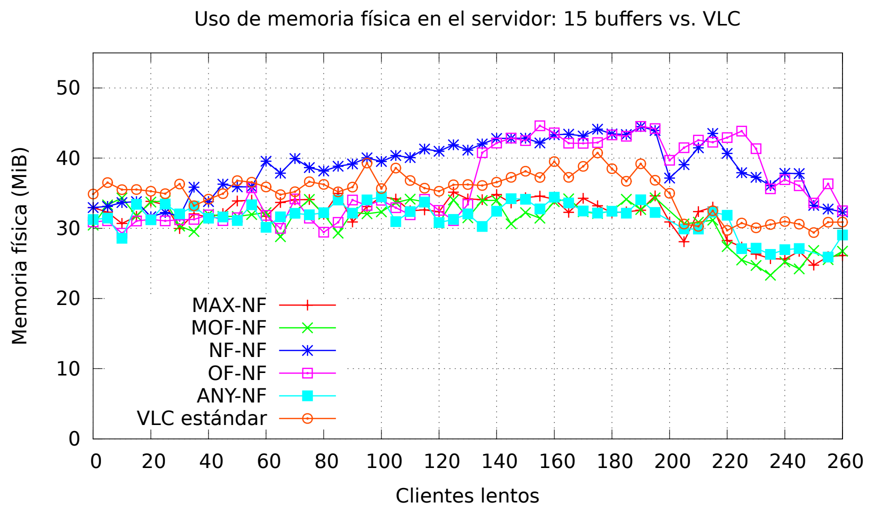


Figura 4.6: Mediciones de uso de memoria física con 2 clientes rápidos, usando 6, 8 y 10 buffers.





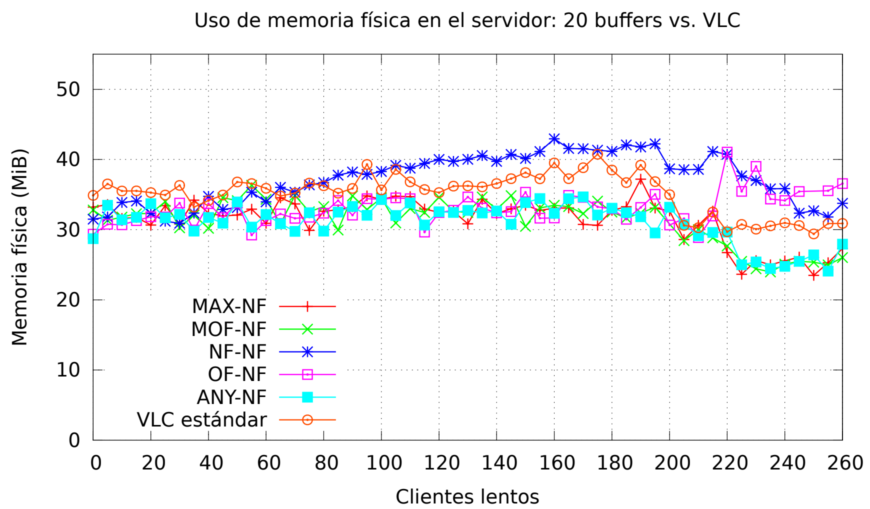


Figura 4.7: Mediciones de uso de memoria física con 2 clientes rápidos, usando 12, 15 y 20 buffers.

Uso de memoria virtual

En la Figura 4.8 se muestra el uso de memoria virtual. Solo se incluye el gráfico para el algoritmo MAX, ya que los otros algoritmos mostraron un comportamiento idéntico.

Se observa que en VLC sin algoritmos se mantiene entre 180 MiB y 190 MiB en todas las pruebas, mientras que al usar algoritmos crece de manera proporcional al número de clientes lentos conectados. A cada *thread* de ejecución se reserva un espacio de memoria al momento de ser iniciado, pero ese espacio no es usado en memoria física hasta que el *thread* efectivamente escribe datos sobre ese espacio. Por esta razón el uso de memoria física no crece de la misma manera que el uso de memoria virtual, ya que los *threads de cliente* no utilizan toda la memoria que tienen reservada. Se observa además otra evidencia de la administración de memoria que hace el sistema operativo: en este caso sí aumenta el uso de memoria virtual a medida que se agregan buffers, pero como se observa en los gráficos de uso de memoria física esto no se traduce en un uso efectivo de esa memoria.

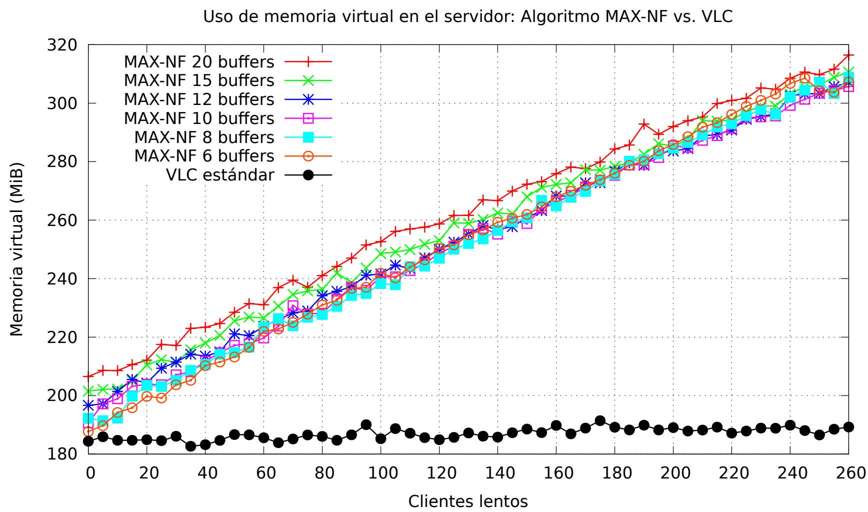


Figura 4.8: Mediciones de uso de memoria virtual con dos clientes rápidos, usando el algoritmo de asignación MAX y el algoritmo de reemplazo NF.

4.1.2. Memoria virtual en *threads de cliente*

Para verificar que el aumento en el uso de memoria virtual que se observa en la Figura 4.8 se debe principalmente a la memoria reservada para los *threads de cliente*, se hace una prueba similar pero disminuyendo el tamaño del *stack* de memoria que se reserva a cada *thread* al momento de su creación.

Se usa el algoritmo de asignación MAX y el algoritmo de reemplazo NF con 8 buffers, y se ejecutan tres pruebas bajo las mismas condiciones que en la prueba descrita en la sección 4.1.1.

La Figura 4.9 presenta el uso de memoria virtual que se mide al usar 256 KiB, 128 KiB y 64 KiB de *stack*. Se incluye también el resultado al usar 512 KiB, que es el tamaño de *stack* que usa VLC por defecto. Se observa que, si bien el uso de memoria virtual igualmente crece

a medida que se agregan clientes, este crecimiento es menos pronunciado a medida que se reduce el tamaño del *stack*.

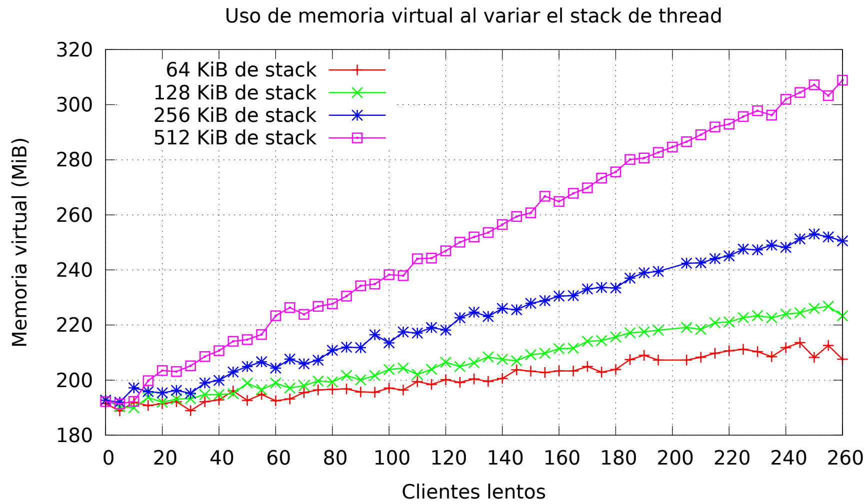


Figura 4.9: Uso de memoria virtual, usando 64 kiB, 128 KiB y 256 KiB para el stack de thread.

4.1.3. Errores de decodificación

Para esta prueba se ejecuta el servidor y se conecta un único cliente lento a 150 KiB/s, modificado para contabilizar la cantidad de errores que se producen durante el proceso de decodificación. Se mide además el FPS del cliente desde el servidor y desde el cliente. El servidor corre durante cinco minutos, y se repite esta prueba 10 veces para verificar que los resultados son consistentes. Este proceso se hace primero usando VLC sin algoritmos, y luego usando el algoritmo de asignación MAX y el algoritmo de reemplazo NF.

En el Cuadro 4.1 se muestran los resultados de esta prueba al usar algoritmos. Se observa que en ninguna prueba se producen errores de decodificación en el cliente, y las medidas de FPS desde el cliente y desde el servidor son muy similares.

Prueba	Errores de decodificación	Errores /minuto	FPS medido en el cliente	FPS medido desde el servidor
1	0	0.00	1.45	1.47
2	0	0.00	1.48	1.51
3	0	0.00	1.46	1.49
4	0	0.00	1.47	1.50
5	0	0.00	1.47	1.50
6	0	0.00	1.46	1.49
7	0	0.00	1.47	1.49
8	0	0.00	1.46	1.49
9	0	0.00	1.47	1.49
10	0	0.00	1.47	1.51

Cuadro 4.1: Errores de decodificación en un cliente lento al usar algoritmos

En el Cuadro 4.2 se muestran los resultados de la prueba sin usar algoritmos. Se observa que se producen múltiples errores de decodificación, y que la medición de FPS difiere entre el servidor y el cliente.

Prueba	Errores de decodificación	Errores /minuto	FPS medido en el cliente	FPS medido desde el servidor
1	150	29.41	0.72	1.24
2	150	29.32	0.74	1.26
3	146	28.53	0.70	1.21
4	150	29.22	0.72	1.25
5	151	29.42	0.70	1.23
6	148	28.93	0.75	1.26
7	150	29.32	0.69	1.22
8	152	29.71	0.72	1.25
9	150	29.32	0.76	1.29
10	151	29.51	0.67	1.20

Cuadro 4.2: Errores de decodificación en un cliente lento sin usar algoritmos

Esto sucede porque VLC define un buffer de tamaño fijo para cada cliente. Al momento de obtener datos desde el buffer del stream para enviarlos al cliente, si un frame completo no cabe en el buffer del cliente, solo se envía la porción del frame que sí cabe en el buffer, y en la siguiente llamada a la función **ClientSend** se continúa enviando los datos que falten. Si el cliente es demasiado lento, es probable que su posición en el buffer principal del stream sea sobreescrita entre una llamada a **ClientSend** y la siguiente, y el resultado es que el cliente pierde la porción de datos de ese frame que no alcanzó a enviar. Los errores de decodificación se producen porque VLC está enviando frames incompletos a los clientes lentos, y es importante destacar que estos errores se producen aún cuando se usa un protocolo de transmisión confiable como TCP, es decir, no son producto de pérdida de paquetes durante la transmisión. Por estas razones el FPS del cliente disminuye, ya que está recibiendo frames incompletos que no puede mostrar, y la medición de FPS desde el servidor es mayor que en el cliente porque el servidor envía datos que el cliente no puede utilizar.

4.1.4. Rendimiento de VLC estándar

En las resultados presentados en la Sección 4.1.1 se observa que el rendimiento de VLC estándar comienza a degradarse a partir de los 140 clientes conectados. VLC solo utiliza dos threads al hacer streaming: uno para escribir datos sobre el buffer, y otro para atender a todos los clientes. La razón del deterioro en el rendimiento de VLC estándar, entonces, no debe ser la sincronización entre estos threads al leer y escribir desde el buffer, como sucede al usar los algoritmos, y debería ser una consecuencia del uso ineficiente de CPU. En teoría, mientras más CPU tenga disponible el servidor, más clientes podrán conectarse antes de que se vea una disminución en el rendimiento.

Para comprobar esto se realiza una prueba adicional con VLC estándar, esta vez con clientes rápidos que no muestran el video que reciben, liberando así recursos para el servidor VLC (ya que el servidor y los clientes se ejecutan en la misma máquina). Cuando un cliente

no muestra la ventana de video requiere menos tiempo de CPU en su ejecución, pero no es posible medir desde el cliente el FPS que recibe. Por esta razón se utiliza la medición de FPS que se obtiene desde el servidor.

Los resultados de esta prueba se presentan en la Figura 4.10.

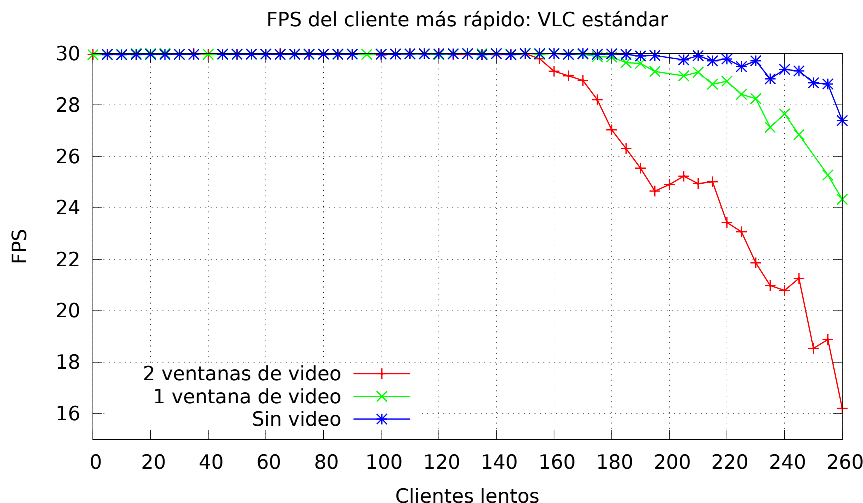


Figura 4.10: Medición de FPS en el cliente más rápido en VLC estándar, variando el número de clientes cuyo video recibido se muestra en pantalla.

Se observa que el rendimiento de VLC comienza a disminuir a partir de los 220 clientes cuando ningún cliente rápido muestra video, y a partir de los 180 clientes cuando un solo cliente rápido muestra el video que recibe. Al comparar esto con las pruebas antes mencionadas, en que los dos clientes rápidos muestran video y la disminución de FPS comienza a partir de los 140 clientes, se puede ver que siempre se encuentra un límite en que el rendimiento en VLC comienza a disminuir. Este límite se puede superar usando los algoritmos de asignación y reemplazo de frames y una cantidad suficiente de buffers, utilizando el poder de procesamiento disponible de manera más eficiente.

4.2. Alternativa 2

En las pruebas con la Alternativa 2, dado que el video se codifica una vez por cada cliente conectado, se transmite el mismo video utilizado en las pruebas de la Sección 4.1 pero reducido a una resolución de 320x180, con el objetivo de disminuir la demanda de recursos durante la ejecución del servidor. La codificación del video se hace con el formato MPEG-1, que utiliza compresión *interframe* y por lo tanto genera I-frames, P-frames y B-frames.

4.2.1. Dos clientes rápidos

En esta prueba se conectan múltiples clientes lentos y se mide el FPS en dos clientes rápidos, que pueden recibir video a la máxima velocidad posible. Los clientes lentos tienen velocidades distintas en el rango 34-36 KiB/s que se asignan aleatoriamente, recibiendo video en el rango 0.5-1.5 FPS, asemejando las condiciones estudiadas en el paper.

En cada prueba individual se conecta una cantidad definida de clientes lentos, que va desde 0 hasta 100. Por cada una se ejecuta el servidor y se conectan clientes cada 0.5 segundos. Se hacen las mediciones de FPS, uso de memoria y uso de CPU durante 5 minutos desde el momento en que se conecta el último cliente.

FPS

En la Figura 4.11 se muestran los resultados de las mediciones de FPS utilizando el algoritmo de asignación MAX y el algoritmo de reemplazo NF. Se observa que a partir de los 20 clientes conectados el rendimiento del cliente más rápido disminuye drásticamente, aún en las pruebas en que se dispone un número mayor de buffers. Este comportamiento se observa con todos los otros algoritmos de asignación al usar la Alternativa 2 .Incluso se observa cuando se aumenta la velocidad de los clientes lentos, lo que en teoría debe mejorar el rendimiento (ya que la transmisión de cada frame toma menos tiempo y los buffers se liberan para ser sobreescritos más rápidamente, evitando pérdidas).

La razón principal de este deterioro es el intensivo trabajo que debe realizar el servidor al tener que hacer la codificación cada vez que se envía un frame a un cliente, destinando más tiempo de procesamiento a la codificación que a la escritura de nuevos frames en el pool. El resultado es que se pierden frames en el servidor, aún cuando hay buffers disponibles.

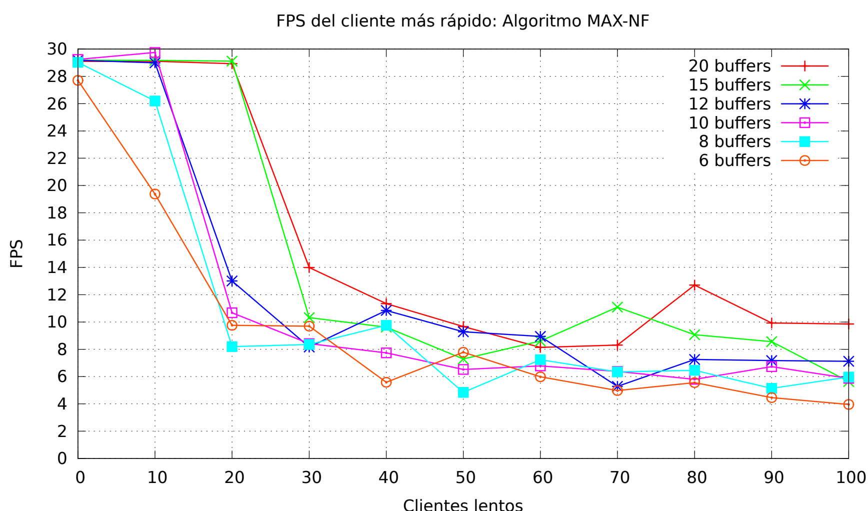


Figura 4.11: Medición de FPS en el cliente más rápido, utilizando los algoritmos MAX-NF.

Uso de CPU

La Figura 4.12 muestra el uso de CPU medido en el servidor durante esta prueba. Se observa que el uso de CPU aumenta a medida que se conectan más clientes, pero a diferencia de lo observado con la Alternativa 1, esto no se traduce en un aumento de rendimiento para los clientes rápidos.

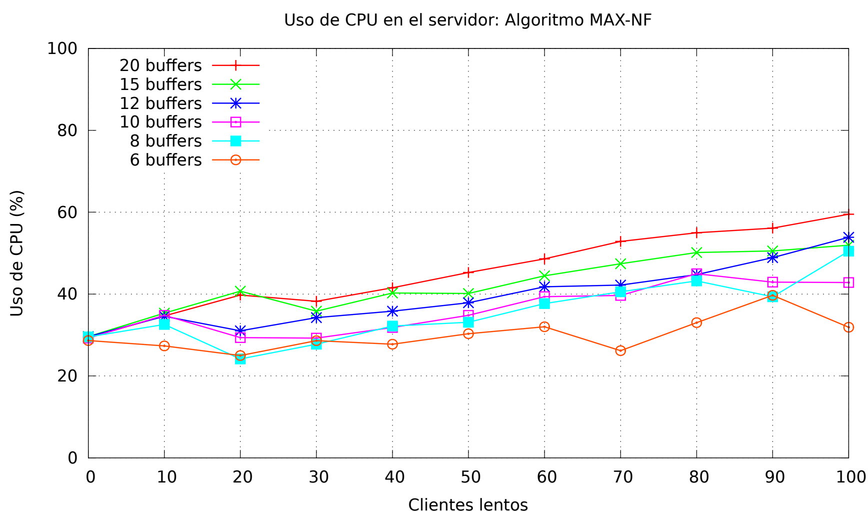


Figura 4.12: Mediciones de uso de CPU con 2 clientes rápidos, utilizando los algoritmos MAX-NF.

Uso de memoria física

En la Figura 4.13 se muestran las mediciones de uso de memoria física en el servidor. Se observa un crecimiento proporcional al número de clientes conectados que es esperable. Como se menciona en la Sección 3.3.2, se debe crear y guardar un *contexto de codificación* por cada cliente para hacer la codificación de cada frame enviado, y este contexto se debe mantener en memoria mientras el cliente esté conectado.

Uso de memoria virtual

La Figura 4.14 muestra el uso de memoria virtual en el servidor. Se observa que, a medida que se conectan clientes, el uso de memoria virtual crece proporcionalmente. Esto responde a dos factores: la memoria reservada a cada thread de cliente (que no necesariamente corresponde a memoria efectivamente usada), y a la memoria reservada y utilizada por el *contexto de codificación* de cada cliente.

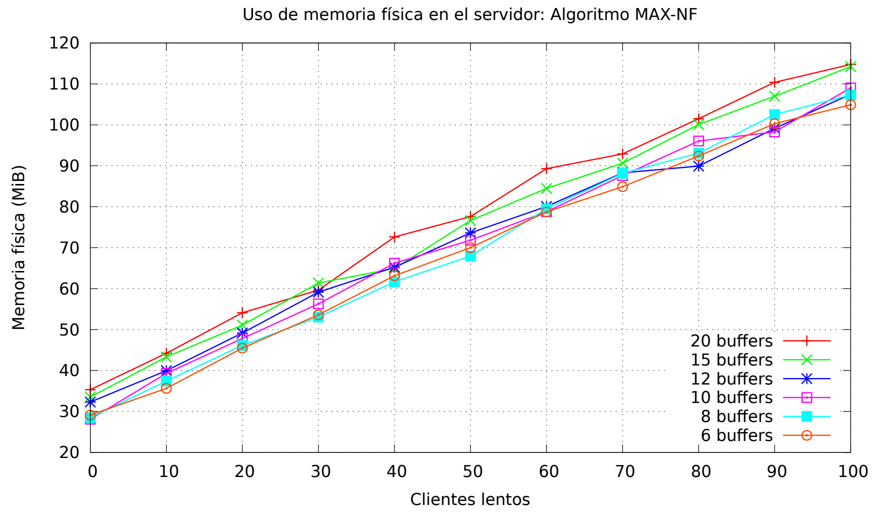


Figura 4.13: Mediciones de uso de memoria física con 2 clientes rápidos, utilizando los algoritmos MAX-NF.

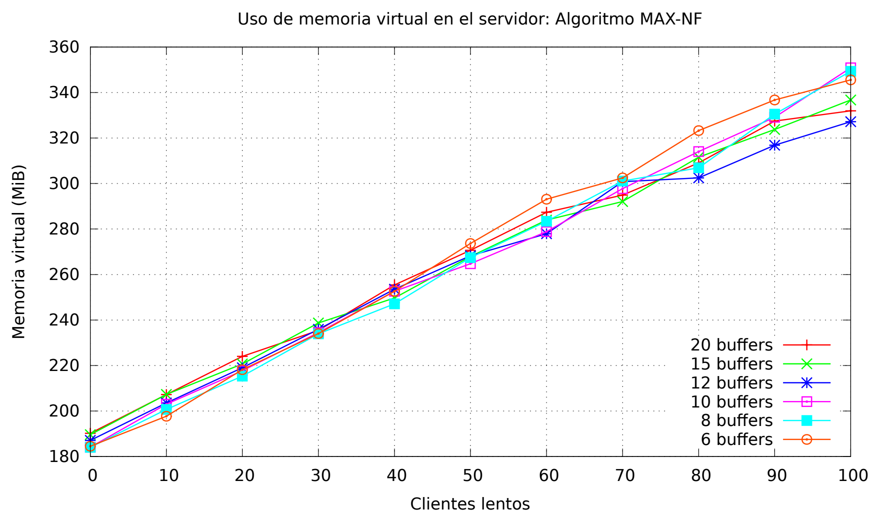


Figura 4.14: Mediciones de uso de memoria virtual con 2 clientes rápidos, utilizando los algoritmos MAX-NF.

4.2.2. Errores de decodificación

Para esta prueba se ejecuta el servidor y se conecta un único cliente lento a 35 KiB/s, modificado para contabilizar la cantidad de errores que se producen durante el proceso de decodificación. Se mide además el FPS del cliente. El servidor corre durante cinco minutos, y se repite esta prueba 10 veces para verificar que los resultados son consistentes.

En el Cuadro 4.3 se muestran los resultados de esta prueba. No se producen errores de decodificación en el cliente lento, lo que permite verificar que la codificación que se hace en el servidor para cada cliente evita los errores que se producirían al usar la Alternativa 1 de implementación con un formato de video como MPEG-1 que utiliza compresión *interframe*. Con la Alternativa 2, los clientes siempre reciben todos los frames de referencia que necesitan para decodificar P-frames y B-frames, permitiendo que el servidor descarte frames sin que esto produzca errores en los clientes.

Prueba	Errores de decodificación	FPS medido en el cliente
1	0	1.37
2	0	1.23
3	0	1.47
4	0	1.51
5	0	1.44
6	0	1.43
7	0	1.39
8	0	1.32
9	0	1.34
10	0	1.41

Cuadro 4.3: Errores de decodificación en un cliente lento al usar algoritmos

Capítulo 5

Conclusiones

A través del estudio y comprensión del funcionamiento interno de VLC se implementaron los algoritmos de asignación y reemplazo de frames, permitiendo su uso para hacer streaming de video desde fuentes reales hacia clientes reales. Se realizaron pruebas para medir el funcionamiento e impacto de los algoritmos en VLC, y para comparar su desempeño con el funcionamiento normal de VLC.

La conclusión más importante de este trabajo es que efectivamente se logra una mejora notoria en el desempeño del servidor VLC y de los clientes conectados al usar una combinación de algoritmos adecuada y un número de buffers suficiente. El servidor logra atender a múltiples clientes concurrentes de manera efectiva, usando menos tiempo de CPU y manteniendo el uso de memoria dentro de un rango acotado, generando mejores resultados en los clientes usando los algoritmos MAX, MOF o ANY y cuando se dispone de 12 o más buffers.

Durante el análisis de las mediciones que se hicieron del uso de memoria se observó que, a diferencia de VLC estándar, hay un crecimiento pronunciado en el uso de memoria virtual al usar los algoritmos, y que es proporcional al número de clientes conectados. Se pudo comprobar, sin embargo, que este crecimiento corresponde solo a la memoria que cada thread de cliente tiene reservada, y en las mediciones de uso de memoria física es posible ver que solo una porción acotada de esa memoria se usa efectivamente.

Un comportamiento que se observó en las pruebas con VLC es que el algoritmo estándar que se usa para hacer streaming envía frames incompletos a los clientes lentos, generando errores de decodificación, y este comportamiento no depende del protocolo de transmisión de red que se utilice para establecer la conexión. Se comprobó a través de las pruebas realizadas que estos errores no se producen al usar los algoritmos de asignación y reemplazo de frames, ya que se evita que en el servidor se sobreescriba un frame mientras un cliente no lo haya recibido en su totalidad. Si bien esto tiene como consecuencia que el servidor tiene menos memoria disponible para escribir nuevos frames, es posible compensar utilizando un pool de buffers más grande sin que el uso de memoria física crezca demasiado, obteniendo así un mejor rendimiento para clientes lentos y rápidos.

El deterioro en el rendimiento de VLC para los clientes rápidos no se vio dentro del rango de los 100 clientes conectados, que es el rango en que se realizaron las pruebas originales en el paper. Este deterioro sí se observa cuando se superan los 140 clientes conectados, y se

logró comprobar que se produce por la forma en que VLC atiende clientes y no por la cantidad de memoria que utiliza en el proceso. En este sentido se puede concluir que siempre existirá un límite en el rendimiento de VLC, que depende del poder de procesamiento disponible en el servidor, que puede ser mejorado usando los algoritmos de asignación y reemplazo de frames y una cantidad suficiente de buffers.

Si bien se cumplieron los objetivos trazados, durante el proceso de implementación de los algoritmos en VLC se evidenciaron ciertos límites en el alcance del trabajo. Los algoritmos funcionan correctamente con todos los codecs que soporta VLC, pero la limitación más importante fue la necesidad de utilizar solo compresión *intraframe* para obtener video de buena calidad y sin errores al momento de hacer las pruebas. Se pudo proveer una implementación alternativa que soluciona este problema, sin embargo en su funcionamiento es una implementación mucho más demandante en términos de uso de recursos en el servidor, lo que produce un deterioro muy notorio en el rendimiento de los clientes rápidos aún cuando se usan los mejores algoritmos de asignación y con un mayor número de buffers.

En relación a lo anterior, el uso de los algoritmos de asignación y reemplazo de frames provee una mejora significativa con respecto a VLC estándar cuando se requiere o prefiere utilizar solo compresión *intraframe*, por ejemplo cuando el recurso más escaso es el poder de procesamiento y no la velocidad de la conexión. Esto se debe a que, en su estado actual, el uso de los algoritmos no provee un soporte adecuado para la compresión de video *interframe*, que es mucho más efectiva y permite reducir considerablemente la cantidad de datos que se deben transmitir desde el servidor al cliente.

Se identifican múltiples aspectos sobre los que se podría continuar el trabajo presentado. El más importante es mejorar el soporte para la compresión de video *interframe*, particularmente para que se permita aprovechar todas las ventajas que ofrece el formato H.264, uno de los formatos de video más populares en la actualidad [15]. Una posibilidad que se exploró durante la realización de este trabajo es incorporar el tipo de frame como una de las variables que se usan para la selección de frames en los algoritmos de asignación y reemplazo. Así, por ejemplo, un algoritmo de reemplazo podría evitar reemplazar I-frames con P-frames o B-frames, de este modo manteniendo en memoria un conjunto de frames de referencia, y disponer por otro lado de un algoritmo de asignación que prefiera enviar los I-frames que un cliente aún no ha recibido, permitiendo que ese cliente pueda decodificar el stream de mejor manera.

Otro aspecto importante para un trabajo futuro es estudiar el impacto de los algoritmos en la percepción que tienen los usuarios de la calidad del video que reciben. Además de las mediciones empíricas que se usan en este trabajo (uso de CPU, memoria y FPS) es posible incorporar otras métricas, como el tiempo de *buffering* que toma un video al iniciarse, la frecuencia con que es necesario hacer *buffering* durante la transmisión [16] o la calidad del video y su degradación con respecto al material original [17].

Bibliografía

[1] Piquer, J., Bustos-Jiménez, J.: *Frame Allocation Algorithms for Multi-threaded Network Cameras* (2010).

[2] Wurster, L.: *Open Source Software Hits a Strategic Tipping Point*, Harvard Business Review (2011). Consultado el 22 de septiembre de 2013.
http://blogs.hbr.org/cs/2011/03/open_source_software_hits_a_st.html

[3] *Gartner Survey Reveals More than Half of Respondents Have Adopted Open-Source Software Solutions as Part of IT Strategy*, Gartner (2011). Consultado el 22 de septiembre de 2013.
http://www.gartner.com/it/page.jsp?id=1541414

[4] Delio, M.: *Linux: Fewer Bugs Than Rivals*, Wired (2010). Consultado el 22 de septiembre de 2013.
http://www.wired.com/software/coolapps/news/2004/12/66022

[5] *Introduction to VLC*, VideoLAN Wiki (2012). Consultado el 22 de septiembre de 2013.
http://wiki.videolan.org/Documentation:Play_HowTo/Introduction_to_VLC

[6] *VLC media player*, Wikipedia (2012). Consultado el 22 de septiembre de 2013.
http://en.wikipedia.org/wiki/VLC_media_player

[7] *Introduction to the VLC core*, VideoLAN Wiki (2011). Consultado el 22 de septiembre de 2013.
http://wiki.videolan.org/Documentation:Hacker's_Guide/Core

[8] *Documentation: VLC modules loading*, VideoLAN Wiki (2011). Consultado el 22 de septiembre de 2013.
http://wiki.videolan.org/Documentation:VLC_Modules_Loading

[9] *RFC 793: Tranmission Control Protocol (TCP)*, Internet Engineering Task Force (septiembre, 1981). Consultado el 22 de septiembre de 2013.
http://tools.ietf.org/html/rfc793

[10] Allman, M., Paxson, V., Blanton, E.: *RFC 5681: TCP Congestion Control*, Internet Engineering Task Force (2009). Consultado el 22 de septiembre de 2013.
http://tools.ietf.org/html/rfc5681

[11] Postel, J.: *RFC 768: User Datagram Protocol (UDP)*, Internet Engineering Task Force (agosto, 1980). Consultado el 22 de septiembre de 2013.
http://tools.ietf.org/html/rfc768

[12] Winkler, S., Van Den Branden Lambrecht, C. J., Kunt, M.: *Vision and Video: Models and Applications* (2001).

[13] Apostolopoulos, J., Tan, W., Wee, S.: *Video Streaming: Concepts, Algorithms, and Systems* (2002).

[14] Warner, J. C.: *Ubuntu Manpage: top - display Linux tasks*, Ubuntu manuals. Consultado el 22 de septiembre de 2013.
`http://manpages.ubuntu.com/manpages/oneiric/en/man1/top.1.html`

[15] Sinton, F.: *HTML5 Based Video Availability*, MeFeedia (2011). Consultado el 22 de septiembre de 2013.
`http://blog.mefeedia.com/html5-dec-2011`

[16] Dobrian, F. et al.: *Understanding the Impact of Video Quality on User Engagement* (2011).

[17] Zoran, M., Zoran, B.: *Subjective Video Quality Assessment in the H.264/AVC Video Coding Standard* (2012).