UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA

Simulador de Sistema de Streaming Adaptativo basado en HTTP

TALLER DE SISTEMAS CIBER-FISICOS

Montevideo, 28 de julio de 2022

Micaela Larraura
Martín Rotti
Sofía Tito Virgilio

Tutores: Javier Baliosian, Matías Richart

Índice

1. Introducción

En este documento se detallan el diseño y la implementación de un simulador de un sistema de Streaming Adaptativo basado en HTTP (HAS), incluyendo tanto el cliente como el servidor de dicho sistema.

En la sección 2 introducimos la motivación y el problema a resolver, detallando el objetivo a alcanzar y describiendo el funcionamiento de un sistema HAS.

Luego, en la sección 3 presentamos la solución propuesta, que consiste en el diseño de un prototipo de un simulador de un sistema HAS con múltiples clientes, múltiples servidores y un balanceador de carga que gestiona la comunicación entre clientes y servidores. Además, describimos una implementación de una versión simplificada del prototipo, con múltiples clientes y un único servidor, detallando los parámetros configurables del simulador desarrollado. En esta sección también documentamos las dificultades técnicas más relevantes contra las cuales nos enfrentamos.

Además, en la sección 4 presentamos las métricas y la validación funcional realizada para evaluar el funcionamiento del simulador, junto con la interpretación de los resultados obtenidos. A su vez, se incluye un plan de validación para evaluar su desempeño.

Por otro lado, en la sección 5 detallamos algunas limitaciones del sistema desarrollado, así como también pasos a seguir para complementar o profundizar el trabajo realizado.

Finalmente, en la sección 6 presentamos las conclusiones sobre el trabajo realizado y las características del producto obtenido.

2. Problema

El negocio de las telecomunicaciones está evolucionando hacia la oferta de un conjunto cada vez más rico de servicios. Por esto mismo se tiene la necesidad de generar entornos de ejecución versátiles, capaces de ejecutar diferentes cargas de trabajo en diferentes momentos, y al mismo tiempo desperdiciar la menor cantidad de recursos posible. Esta adaptabilidad se conoce como elasticidad. Para ello se han propuesto muchos controladores basados en Deep Neural Network (DNN) y aprendizaje por refuerzo para entornos similares, sin embargo estas técnicas necesitan un período de aprendizaje que podría no ser accesible para algunos operadores. Por lo anteriormente descrito, se buscan maneras de aplicar técnicas de transfer learning para acelerar el proceso de aprendizaje, realizándolo en un simulador y transfiriendo la política aprendida al sistema real, reduciendo de esta manera los costos de empezar a aprender desde cero en el sistema real.

En este contexto es fundamental contar con un simulador que refleje las características del sistema relevantes para el proceso de aprendizaje, en particular, que permita predecir su performance en base a los recursos disponibles, sin perder de vista la eficiencia en términos de tiempo de ejecución que será relevante para su posterior uso en el entrenamiento del algoritmo.

2.1. Objetivo

En el marco del taller el objetivo es diseñar e implementar un simulador de un Sistema de Streaming Adaptativo basado en HTTP, lo cual incluye el diseño y la implementación tanto del cliente como del servidor de dicho sistema.

2.2. Sistema de Streaming Adaptativo Basado en HTTP

El Sistema de Streaming Adaptativo Basado en HTTP (HAS) es una tecnología diseñada para entregar video de la manera más eficiente posible y con la mejor calidad utilizable para cada usuario y dispositivo específico. El contenido de video se codifica en distintas representaciones que difieren en sus niveles de calidad y cada una de ellas es particionada en segmentos que típicamente contienen de 2 a 10 segundos de video. La adaptación es posible cambiando el frame rate, la resolución o el formato del video, lo que se puede hacer con varias estrategias de adaptación y acciones relacionadas del lado del cliente y del servidor.

A medida que los segmentos que llegan al cliente se van reproduciendo estos se van concatenando para obtener el video completo, pudiendo tener diferentes representaciones para distintos segmentos. Por lo tanto, cambiar la representación es factible en el límite de cada segmento.

Se cuenta con un buffer encargado de almacenar los próximos segmentos a reproducirse, el cual debe mantenerse con una cantidad de segmentos lo suficientemente grande para mitigar el impacto de caídas de rendimiento y pérdidas de enlaces. De esta manera se logra evitar interrupciones en el video que está siendo reproducido.

Un cliente HAS inicia una nueva sesión descargando un archivo de manifiesto. Este archivo de manifiesto proporciona una descripción de los diferentes niveles de calidad y segmentos disponibles. En el núcleo del cliente se encuentra un algoritmo de adaptación que se encarga de seleccionar la representación del próximo segmento a descargar según las condiciones de la red y el nivel actual de llenado del buffer.

Para los usuarios finales, los principales beneficios de HAS en comparación con la transmisión de video HTTP clásica son la reducción de las interrupciones de la reproducción de video y una mejor utilización del ancho de banda, que generalmente dan como resultado una QoE más alta.

Al ejecutarse sobre HTTP, HAS reutiliza la infraestructura ya implementada, como servidores HTTP, proxies HTTP y redes de distribución de contenido (CDN). Debido a estas ventajas, los principales actores (como Microsoft, Apple, Adobe y Netflix) adoptaron este paradigma de transmisión. Como la mayoría de las soluciones HAS usan la misma arquitectura, el Motion Picture Expert Group (MPEG) propuso un estándar llamado Dynamic Adaptive Streaming over HTTP (DASH).

3. Solución Propuesta

La solución propuesta consiste en el diseño de un prototipo del sistema a simular y la implementación de una versión simplificada del mismo.

3.1. Diseño

En la figura 1 podemos observar un prototipo general de un sistema HAS con múltiples clientes, múltiples servidores y un balanceador de carga que funciona como intermediario y sigue alguna política de asignación de clientes a servidores a definir, como podría ser por ejemplo enviar los nuevos clientes al servidor con menor carga de trabajo.

En las secciones 3.1.1 y 3.1.2 explicamos en detalle el comportamiento de un cliente y un servidor del sistema, respectivamente.

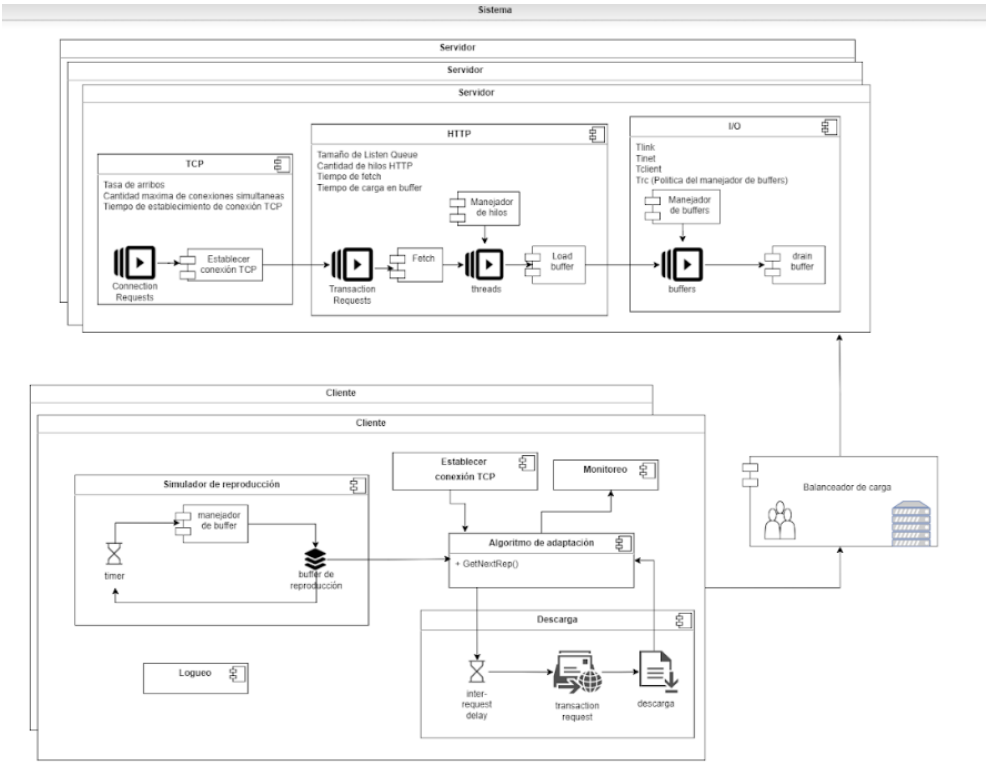

Figura 1: prototipo de un sistema HAS

3.1.1. Cliente

El objetivo de un cliente HAS es descargar y reproducir un vídeo ofrecido por el servidor, para lo cual debe descargar y reproducir cada segmento de vídeo, eligiendo la representación adecuada en cada caso. Para ello cuenta con:

- Algoritmo adaptativo: encargado de monitorear el estado actual y decidir la representación del próximo segmento a solicitar y si corresponde o no insertar un tiempo de espera antes de solicitarlo (inter-request delay). Este algoritmo es invocado cada vez que se va a solicitar un segmento de vídeo al servidor.
- Buffer de reproducción: encargado de almacenar los segmentos de vídeo descargados que aún no fueron reproducidos.

El diseño del cliente está basado en la máquina de estados descrita por Ott et al. [1]. Inicialmente el cliente establece la conexión con el servidor, una vez que la conexión está establecida, invoca al algoritmo adaptativo para que le indique la representación del próximo segmento a solicitar y el inter-request delay correspondiente. Dada esta información el cliente espera (si corresponde) y luego envía la solicitud del segmento al servidor. Una vez que el segmento solicitado llega al cliente, este es depositado en el buffer de reproducción y se invoca al algoritmo adaptativo para solicitar el próximo segmento. Luego de que el primer segmento es depositado en el buffer de reproducción, se comienza a simular la reproducción del vídeo, para esto se toma un segmento del buffer de reproducción y se simula su reproducción en base a su duración. Cuando este segmento termina de reproducirse, se pasa al siguiente segmento del buffer, o, si este esta vacío, se invoca al algoritmo adaptativo para realizar una nueva solicitud.

Notar que se realizan nuevas solicitudes siempre que se completa la descarga de un segmento previamente solicitado o se vacía el buffer de reproducción.

Además cuenta con componentes de monitoreo y logueo, donde el primero se encarga de monitorear métricas utilizadas por el algoritmo adaptativo (como por ejemplo el nivel del buffer de reproducción) y el segundo se encarga de registrar métricas relevantes para el posterior análisis del sistema.

3.1.2. Servidor

El objetivo del servidor es recibir las solicitudes de transacción de los clientes, identificar y obtener el segmento de vídeo solicitado y entregárselo al cliente, por lo que puede ser modelado como un Servidor Web.

Nos basamos en el modelo propuesto por Van Der Mei et al. [2], en el que cada transacción atraviesa tres etapas: Establecimiento de la conexión TCP, procesamiento HTTP y procesamiento I/O. A continuación detallamos cada una de ellas:

1) Establecimiento de la conexión TCP:
En esta fase se establece una conexión bidireccional entre el cliente y el servidor de manera previa a que se empiecen a transmitir los datos. Para llevar a cabo esta conexión se realiza un procedimiento que consiste en tres pasos:

- El cliente envía una solicitud de conexión al servidor (SYN).
- Al llegarle una solicitud, el servidor le envía al cliente un mensaje de reconocimiento (SYN-ACK).
- El cliente recibe el mensaje de reconocimiento y envía un mensaje para confirmar el establecimiento de la conexión (ACK).

El subsistema TCP consta de una cola de espera atendida por un *server daemon* que puede atender un número finito de solicitudes simultáneas. Si una solicitud de conexión llega al servidor y este está completamente ocupado, el mismo rechaza la conexión, teniendo el cliente que iniciar una nueva solicitud. Una vez superado el tercer paso se establece efectivamente la conexión entre el cliente y el servidor, permitiendo de esta manera al cliente enviar la primera solicitud de transacción. Al recibir esta solicitud, el *server daemon* la reenvía hacia el subsistema HTTP para que la atienda.

2) Procesamiento HTTP:
Luego de establecida la conexión TCP, la solicitud de transacción podrá ser manejada por el subsistema HTTP. La misma entra en una cola de espera, la cual es atendida por varios hilos HTTP. Si la cola se encuentra completa, se rechaza la solicitud de transacción y se cierra la conexión con el cliente. De lo contrario, se espera a que esté disponible un hilo HTTP para que la pueda atender. El hilo se encargará de obtener el segmento de video solicitado y lo colocará en un buffer de entrada/salida. Si todos los buffer se encuentran ocupados, el hilo se quedará esperando hasta que haya alguno disponible. Terminado el proceso de colocación del segmento en el buffer, el hilo se libera y puede atender a otra solicitud. Es importante tener en cuenta que en caso de que el segmento solicitado exceda la capacidad del buffer, el hilo debe particionar el segmento y cargar todas las porciones en el mismo buffer, en particular esto implica que debe esperar a que se realice el drenado de todas las porciones (menos la última) antes de ser liberado.

3) Procesamiento I/O:
Una vez que se completa un buffer, el contenido del mismo deberá ser drenado hacia la red. El proceso de drenado de los buffer se hace en esta última etapa en base a una política de asignación, que consiste en visitar cada buffer siguiendo una política round-robin y drenar una parte de él equivalente a un bloque de archivo. Cuando se termina de drenar completamente un buffer, el mismo se libera y queda disponible para que otro hilo HTTP puede colocar más contenido en él.

3.2. Implementación

La implementación fue realizada utilizando una licencia estudiantil de la versión R2022a de las herramientas MATLAB y Simulink. Esta licencia establece un límite de 1000 bloques para la cantidad de bloques que se pueden incluir en el modelo, siendo este uno de los motivos por los que implementamos una versión simplificada del prototipo descrito en la sección 3.1. Utilizamos esta herramienta porque nos permite implementar el modelo de forma lineal, ya que cuenta con implementaciones de colas, servidores, switches, y otros elementos que nos son de utilidad a la hora de bajar a tierra el prototipo diseñado. También permite trabajar con funciones de manera de poder agregar lógica personalizada a la hora de tomar decisiones. Además, permite incluir parámetros configurables en la implementación, cuyos valores determinan el comportamiento del sistema, pudiendo ser cargados desde un archivo de configuración antes de comenzar la simulación.

En la figura 2 podemos observar la implementación propuesta, que cuenta con un máximo de 10 clientes y un servidor HAS. La cantidad de clientes que envían solicitudes durante la simulación es configurable. Las solicitudes de todos los clientes son redirigidas al servidor y cada cliente tiene un identificador que es utilizado luego para redirigir las respuestas del servidor al cliente correspondiente.

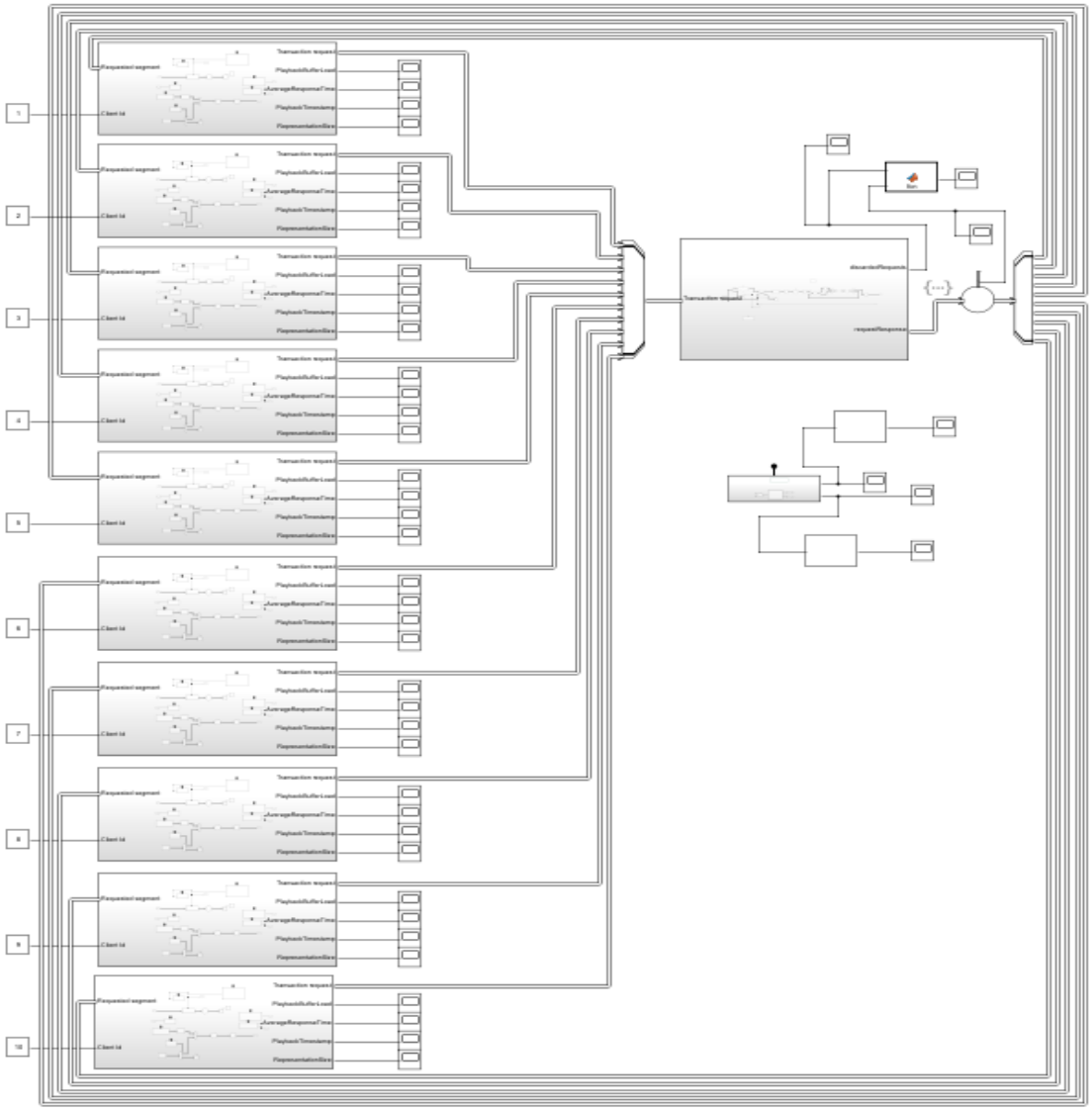

Figura 2: Implementación de un Sistema HAS con un máximo de 10 clientes y 1 servidor

La versión simplificada implementada consiste de un servidor que ofrece un vídeo, segmentado en segmentos de una cierta duración, y codificado en una cierta cantidad de representaciones distintas, donde cada representación tiene un tamaño de segmento asociado. Luego, cada cliente establece una conexión con el servidor y comienza a solicitarle y reproducir segmentos, la representación de cada segmento solicitado queda determinada por el algoritmo adaptativo.

En la sección 3.2.1 detallamos los parámetros configurables del simulador. Luego, en las secciones 3.2.2 y 3.2.3 describimos en detalle la implementación de un cliente y un servidor HAS, respectivamente.

3.2.1. Configuración

La implementación recibe una lista de parámetros que se cargan desde un archivo externo antes de ejecutar la simulación. Los valores de estos parámetros se pueden configurar y son los siguientes:

- *clients*: cantidad de clientes habilitados. Debe ser menor o igual a 10.

- *playback_buffer_capacity*: capacidad (en segmentos) del buffer de reproducción que tiene cada cliente.
- *duration*: duración de un segmento de vídeo en segundos. Todos los segmentos comparten duración.
- *threshold*: threshold que utiliza el algoritmo adaptativo para seleccionar la representación.
- *delay_threshold*: threshold que utiliza el algoritmo adaptativo para definir el inter-request delay.
- *representations*: cantidad de representaciones diferentes de un segmento que guarda el servidor.
- *representation_default*: número natural que identifica a la representación por defecto que se pedirá inicialmente al servidor.
- *representation_sizes*: arreglo que contiene los tamaños de los segmentos asociados a cada representación.
- *simultaneousConnections*: cantidad de establecimientos de conexión simultáneos que permite el server.
- *RTT*: tiempo que toma en establecerse la conexión entre cliente y server.
- *httpQueueCapacity*: capacidad de la cola HTTP (unidades).
- *httpThreads*: cantidad de hilos HTTP del server.
- *fetch*: velocidad del fetch en unidades/segundo.
- *ioBuffers*: cantidad de I/O buffers del server.
- *buffer_capacity*: indica la capacidad de cada I/O buffer (unidades).
- *blockSize*: tamaño de un bloque de archivo (unidades).
- *drainTime*: tiempo en segundos que tarda el servidor en drenar un bloque de un archivo.

3.2.2. Cliente

En la figura 3 podemos observar la implementación de un cliente HAS. Cada cliente recibe 2 entradas, una correspondiente a las respuestas del servidor, que simulan los segmentos descargados, y otra correspondiente al identificador del cliente. Además, tiene una salida correspondiente a las solicitudes enviadas al servidor, y cuatro salidas correspondientes a las métricas: carga del buffer de reproducción, tiempo de respuesta promedio, timestamp de inicio de reproducción de los segmentos y su tamaño.

Tanto las solicitudes como las respuestas se implementan mediante una misma Entidad de tipo Bus, que es generada en el cliente y viaja a través del servidor, volviendo luego al cliente, dando por finalizada la descarga del segmento solicitado.

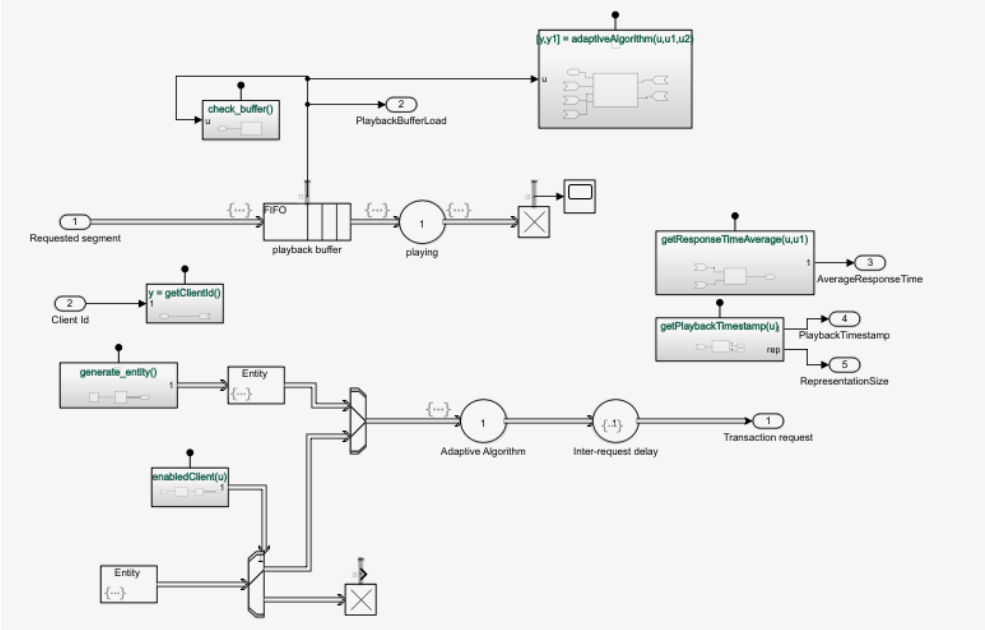

Figura 3: Implementación de un cliente HAS

En la figura 4 se listan y describen los atributos asociados a cada entidad, junto con la unidad o rango en el que están definidos, según corresponda.

Atributos entidad		
Nombre	Descripción	Unidad (*)
fetchingTime	Tiempo que toma fetchear el segmento	segundos
size	Tamaño del segmento	unidades
duration	Duración del segmento	segundos
delay	Tiempo de espera antes de enviar el request del segmento	segundos
first	Flag que indica si es el primer segmento solicitado por el cliente (1- primero, 2- no primero)	[1,2]
last	Flag que marcará la última porción de cada segmento (1 - ultimo, 2 - no ultimo)	[1,2]
id	id que identifica a la entidad en el sistema	natural
clientId	id que identifica al cliente que realizó la solicitud	[1,5]
startRequest	Timestamp en segundos de cuando se manda la request	segundos
endRequest	Timestamp en segundos de cuando llega la respuesta	segundos
sizeLeft	Lo que queda por drenar de un segmento	unidades
lastDrain	Indica si se dreno el ultimo bloque (1 - ultimo, 2- no ultimo)	[1,2]
(*) cuando nos referimos a unidades pueden ser bits, bytes u otro siempre que se utilice en todos lados la misma unidad		

Figura 4: Atributos asociados a cada Entidad

Como podemos observar en la esquina inferior izquierda de la figura 3, las entidades son generadas en el cliente por dos *Entity Generators*, uno que se encarga de realizar la solicitud inicial (seteando el atributo *first* en 1) y otro que se encarga de generar el resto de las solicitudes (seteando el atributo *first* en 2). Es necesario distinguir la solicitud inicial de las restantes porque en la solicitud inicial, además del tiempo de descarga del segmento, debemos contar el tiempo del establecimiento de la conexión TCP. Luego, la generación de las solicitudes posteriores se maneja mediante la invocación de la función *generate_entity()* que se invoca cada vez que ingresa una entidad al buffer de reproducción, o cuando al terminar de reproducir un segmento se va a buscar el siguiente y el buffer de reproducción está vacío. Además, un cliente realizará solicitudes solamente si se encuentra habilitado, controlándose esto con la función *enabledClient(u)*. La función recibe en *u* la cantidad de clientes habilitados configurada y habilitará para realizar solicitudes a

los clientes con id menor o igual que este parámetro.

Los generadores, además de diferenciar el primer segmento de los restantes, setean los atributos *duration*, *id* y *clientId* de las entidades. El atributo *duration* se inicializa con la duración de los segmentos ofrecidos por el servidor, el atributo *id* con el identificador que le asocia el programa a cada entidad, y el atributo *clientId* con el identificador del cliente recibido como entrada.

Luego de generada la entidad, esta es pasada al servidor *Adaptive Algorithm* que simula el comportamiento del algoritmo adaptativo, este servidor invoca a la función *adaptiveAlgorithm()* que recibe la cantidad de segmentos en el buffer de reproducción (n), el threshold utilizado para seleccionar la representación (rt), y finalmente el threshold (dt) y la duración (d) de los segmentos utilizados para definir el inter-request delay. La representación del próximo segmento se define de la siguiente manera:

- si $n < rt/2$: se pasa a la representación anterior (peor calidad)
- si $rt \geq n \geq rt/2$: se mantiene la representación actual
- si $n > rt$: se pasa a la representación siguiente (mejor calidad)

Mientras que el inter-request delay se define de la siguiente manera:

- si $n \leq dt$: inter-request-delay = 0
- si $n > dt$: inter-request-delay = d

El servidor *Adaptive Algorithm* recibe esta información y en base a la representación anterior define la representación actual, seteando el tamaño correspondiente a esa representación en el atributo *size* de la entidad, y el inter-request delay en el atributo *delay*. Luego, el algoritmo adaptativo pasa la entidad al servidor *Inter-request delay*, que se encarga de esperar el tiempo correspondiente antes de enviar la solicitud al servidor, lo cual se logra pasando la entidad a la salida *Transaction request*.

Cuando el cliente recibe la respuesta a su solicitud, esta es depositada en el buffer de reproducción, el cual está implementado mediante una Cola FIFO de capacidad configurable, la cual cada vez que ingresa un nuevo segmento invoca a la función *generate_entity()* para generar una nueva solicitud.

El buffer de reproducción alimenta al servidor *playing* que se encarga de simular la reproducción de cada segmento basado en el atributo *duration* del mismo. Cuando termina de reproducir un segmento este se descarta en un terminador y se invoca a la función *check_buffer()* que verifica la carga del buffer de reproducción, si este está vacío invoca a la función *generate_entity()* para generar una nueva solicitud, de lo contrario comienza a simular la reproducción del siguiente segmento en el buffer.

3.2.3. Servidor

En la figura 5 podemos observar la implementación de un servidor HAS. Recibe una única entrada, que corresponde a las solicitudes enviadas por clientes. La primera salida corresponde a las respuestas a estas solicitudes, y la segunda a la cantidad de solicitudes descartadas por el servidor.

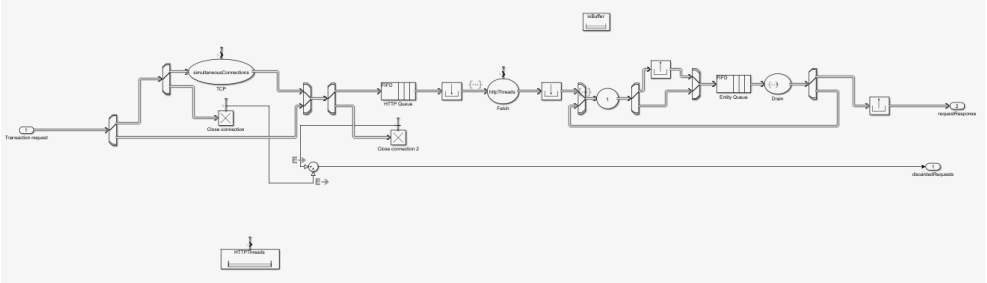

Figura 5: Implementación de servidor HAS

La Entidad de tipo Bus que representa respuestas y solicitudes es la misma que la descrita en la sección 3.2.2.

Como podemos observar en la figura 6, cuando las solicitudes llegan al servidor, lo primero que se hace es identificar si es la primera solicitud de un cliente. Esto se implementa leyendo el valor del atributo *first* de la entidad, seteado en la generación de la misma. Si el valor del atributo *first* es 1, entonces es la primera solicitud para un determinado cliente, y se deberá establecer la conexión TCP antes de continuar.

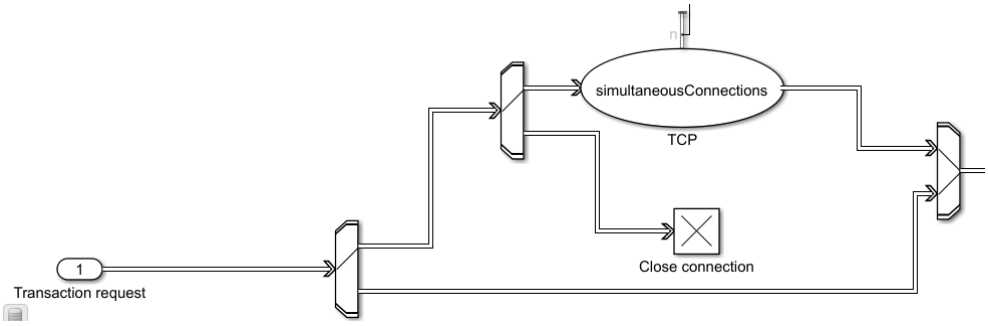

Figura 6: Implementación de establecimiento de conexión TCP

El tiempo en establecer dicha conexión se implementa con el servidor *TCP*, que tiene como capacidad la variable *simultaneaousConections*, y como tiempo de servicio la variable *RTT*, ambas descritas en la sección 3.2.1. En caso de que el servidor *TCP* llegue a su capacidad de entidades, la entidad pasa a un Entity Terminator que representa el cierre de la conexión.

Una vez establecida la conexión TCP (en caso de la primera solicitud), la entidad pasa al procesamiento HTTP, cuya implementación podemos observar parcialmente en la figura 7.

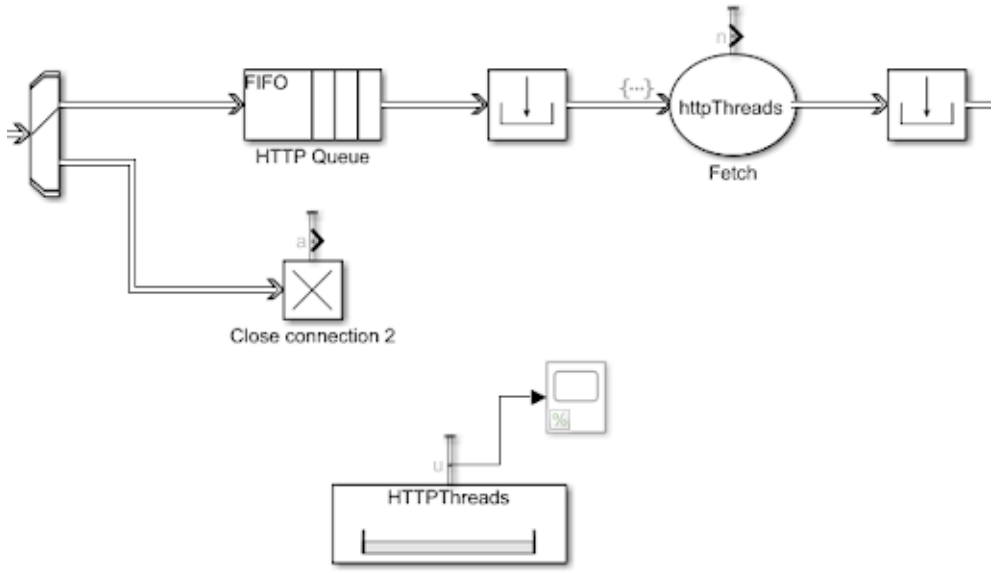

Figura 7: Implementación de procesamiento HTTP

En primer lugar, la entidad pasa a una Cola FIFO *HTTP Queue*, con capacidad *httpQueueCapacity*, y en caso de superar esta capacidad las entidades restantes pasan a un Entity Terminator que representa el cierre de la conexión.

Los hilos HTTP se implementa mediante un Resource Pool *HTTPThreads*, cuya cantidad de recursos está dada por la variable *httpThreads*. Mientras haya hilos HTTP disponibles, las entidades podrán salir de *HTTP Queue* y ocupar un hilo utilizando un Resource Aquirer.

Una vez que una entidad tenga un recurso de *HTTPThreads*, pasa a un servidor *Fetch*. Cuando entra una entidad a este servidor, se actualiza su atributo *fetchingTime* para que sea el valor de su atributo *size* dividido entre la variable *fetch*. Una vez hecho este cambio, se usa el nuevo valor del atributo *fetchingTime* como tiempo de servicio del servidor.

La implementación de los buffers de entrada y salida la encontramos en la figura 8. Como podemos observar, una vez se completó el servicio del servidor *Fetch*, las entidades deben adquirir un recurso para seguir avanzando. Este recurso representa los buffers disponibles, cuya cantidad es indicada por la variable *ioBuffers*. Esto permite que las entidades solo sigan en el caso de que haya un buffer disponible, de lo contrario deberán esperar a que se libere uno.

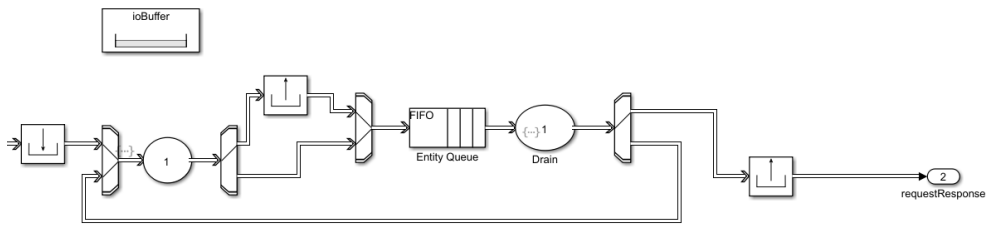

Figura 8: Implementación de procesamiento IO

Las entidades que adquieren el recurso del buffer pasan a un servidor con tiempo de servicio 0, cuyo propósito es identificar la última vez en la que se va a cargar el contenido del segmento al buffer. Esto lo hace comparando el atributo de la entidad, *sizeLeft*, con la capacidad de un buffer, dado por la variable *buffer_capacity*. Si *sizeLeft* <= *buffer_capacity*, entonces lo que queda por cargar en el buffer puede ser cargado por completo, así que le asigna el valor 1 al atributo *last* de la entidad, y en caso contrario le asigna valor 2. Usando este valor es que el simulador puede distinguir entre los dos casos, y usar un switch para enviarlos por distintos caminos. Esto es importante ya que cuando el segmento se carga por completo en el buffer, debe liberar el hilo HTTP, lo cual en la implementación se traduce a liberar el recurso *HTTPThreads*.

Libere o no el hilo HTTP, luego la entidad va a parar a una cola FIFO. Esta cola tiene capacidad *buffer_capacity*, y representa los buffers. Cuando una nueva entidad llega a la cola, sería equivalente a cargar un buffer. Al salir de esta cola, las entidades pasan al servidor *Drain* que tiene capacidad 1, y cuyo tiempo de servicio es la variable *drainTime*. Cuando termina su servicio, le resta al atributo *sizeLeft* de la entidad, la variable *blockSize*, o le setea 0 en caso de que *sizeLeft* sea menor que *blockSize*. Además, en caso de que el nuevo valor de *sizeLeft* sea 0 le seteará valor 1 al atributo *lastDrain* de la entidad, y 2 en caso contrario. Lo que representa este proceso es lo siguiente: De lo que está cargado en el buffer, se drena una cantidad determinada (indicada por *blockSize*), se actualiza lo que queda por drenar de este segmento (indicado por *sizeLeft*), y se pregunta si tiene que seguir drenando o no (indicado por *lastDrain*). Luego, en el siguiente Switch, usando *lastDrain*, se sabe si se debe liberar el recurso del buffer y dejar el Servidor o volver al comienzo para seguir drenando.

3.2.4. Dificultades técnicas

Una de las dificultades técnicas más relevantes a la que nos enfrentamos fue el modelado de los I/O Buffers y la asignación del recurso de drenado a los mismos.

Inicialmente modelamos cada I/O Buffer como una Cola FIFO independiente, en donde la capacidad de la FIFO indica la capacidad de ese I/O Buffer (en cantidad de bloques de archivo que puede almacenar). Esta implementación la podemos observar en la figura 9. En primer lugar se tiene un *Entity Gate* que controla que solamente se pueda depositar un nuevo segmento en el buffer si este está vacío. Luego, cuando una entidad llega a uno de estos buffers, se divide en bloques de archivo de tamaño *blockSize* utilizando un Replicador, que son cargados en el buffer por un hilo HTTP y luego drenados. Cuando se deposita en el buffer la última carga asociada a un segmento, se envía la entidad original por la segunda salida del buffer, que luego pasa por un liberador de recursos, liberando el hilo correspondiente.

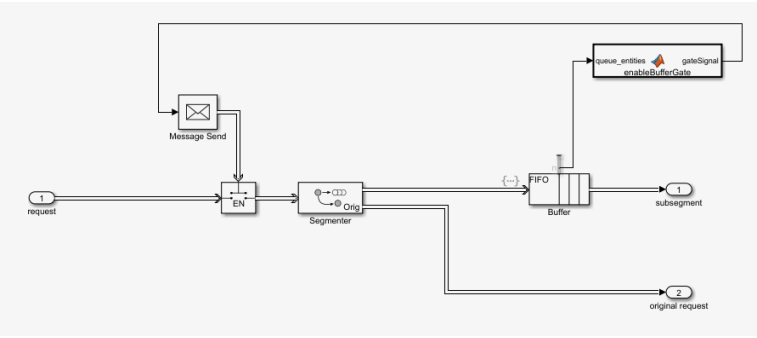

Figura 9: Implementación inicial de un I/O Buffer

Además, manteníamos una cantidad máxima de I/O Buffers, pudiendo seleccionar la cantidad de I/O Buffers a ser utilizados mediante un parámetro, dejando los restantes inutilizados. Esto lo implementamos mediante una serie de switchs anidados en la que se seleccionan los puertos de salida por los que

se envían entidades. Para que fuera escalable implementamos bloques de I/O Buffers anidados, de forma que el selector externo solamente tiene que seleccionar entre 5 bloques de 100 I/O Buffers cada uno, que a su vez están compuestos por 5 bloques de 20 I/O Buffers cada uno (con su propio selector interno). Teniendo entonces un máximo de 500 buffers y una granularidad de 20 en la cantidad de buffers siendo utilizados.

Luego, nos enfrentamos al problema de modelar la política de asignación del recurso de drenado a los distintos I/O Buffers. En el modelo propuesto, se describía una política de asignación round-robin, sin embargo, la política round-robin provista por la herramienta tiene un comportamiento que no es el deseado, ya que al asignarle un turno a un I/O Buffer se queda esperando a que ese I/O Buffer tenga algo para drenar y no avanza hasta que esto no ocurra. Esto representa un problema tanto para los I/O Buffers que nunca serán utilizados como para el tiempo ocioso del recurso de drenado que es desperdiciado a la espera de que llegue algo a un I/O Buffer en particular.

Esto nos llevó a intentar implementar una política round-robin que en caso de asignarle un turno a un I/O Buffer que no tiene nada para drenar, avance al siguiente en lugar de quedarse bloqueado esperando. Esto presentó distintas dificultades, varias de las cuales no pudimos sortear, en particular, enviar mensajes o avisos entre bloques que se encuentran en distintos niveles de anidación, por lo que decidimos descartar la política round-robin y realizar una simplificación en el modelo del servidor, implementando un FIFO entre los buffers que están esperando ser drenados. La implementación de I/O Buffers que teníamos no podía adaptarse directamente a una política FIFO entre buffers, porque podría darse el caso en que varios bloques de archivo de un mismo buffer acapararan el recurso de drenado, por lo que modificamos la implementación de los I/O Buffers llegando a la implementación descrita en la sección 3.2.3.

La implementación propuesta tiene sus ventajas y desventajas. Por un lado, implica una simplificación en el modelo. Sin embargo, es sencillo escalar la cantidad de buffers simplemente editando la variable *buffer_capacity*, que controla tanto la cantidad de recursos de tipo *ioBuffer* como la capacidad de la cola FIFO. A su vez es más eficiente en cuanto al uso de bloques de Simulink, y además este no cambia al aumentar la cantidad de buffers. Con las otras implementaciones que probamos, agregar un solo buffer aumentaba considerablemente la cantidad de bloques, además de implicar un cambio en la lógica. Más aún, nuestra implementación impide que un solo buffer acapare el uso del servidor, ya que una vez drenada parte del segmento, si debe seguir drenando, irá al final de la cola.

4. Verificación

4.1. Métricas

Decidimos registrar las siguientes métricas:

- Cantidad de solicitudes atendidas: cantidad de solicitudes procesadas correctamente por el servidor (segmentos enviados)
- Tasa de error: tasa de solicitudes no procesadas correctamente por el servidor (segmentos solicitados que no se enviaron)
- Tiempo promedio de respuesta: tiempo promedio que transcurre entre que un cliente envía una solicitud y le llega la respuesta correspondiente.
- Throughput: cantidad de solicitudes atendidas (o de unidades transmitidas) sobre tiempo empleado.
- Carga buffer de reproducción: cantidad de segmentos en el buffer de reproducción de cada cliente.

Además, para cada cliente registramos el inicio de reproducción y la calidad de cada segmento solicitado.

Para el registro de estas métricas usamos bloques de tipo *Scope*, que toman una señal y despliegan su valor en función del tiempo.

La cantidad de solicitudes atendidas la calculamos contando la cantidad de entidades que salen del servidor (sA), mientras que para la tasa de error contamos la cantidad de entidades que se descartan (sD) debido a un cierre de conexión por parte del servidor y luego calculamos la tasa como: $tasa\ de\ error = \frac{sD}{sA+sD}$. Para registrar la cantidad de entidades que salen del servidor colocamos un bloque de tipo server a la salida del servidor, y redirigimos su estadística "number of entities departed" hacia un *Scope*. Para registrar la cantidad de entidades que se descartan, sumamos las estadísticas "number of entities arrived" de los bloques *Entity Terminator* que simulan el cierre de conexión y redirigimos la suma hacia un *Scope*.

El tiempo promedio de respuesta lo calculamos utilizando los timestamps *startRequest* y *endRequest* que son atributos de cada entidad que marcan el tiempo de inicio de la solicitud y el tiempo de llegada de la respuesta correspondiente. Al llegar la respuesta al cliente, estos datos se pasan a una función que calcula y actualiza el tiempo de respuesta promedio para ese cliente y redirigimos dicho promedio a un *Scope*. Luego, para obtener el tiempo de respuesta promedio general, debemos calcular un promedio con el tiempo de respuesta promedio para cada cliente.

El throughput lo calculamos de dos formas distintas:

- cantidad de solicitudes procesadas sobre tiempo empleado.
- cantidad de unidades transmitidas sobre tiempo empleado.

Para ello, cada vez que sale una entidad del servidor, se invoca a una función que calcula la cantidad de entidades que salen del servidor y el acumulado del tamaño de los segmentos transmitidos, estas métricas son redirigidas a dos bloques de tipo *Scope* y luego se deben dividir entre el tiempo total de simulación para obtener el throughput correspondiente.

La carga del buffer de reproducción la calculamos graficando la cantidad de entidades que hay en el buffer en función del tiempo, siendo el buffer representado por una Cola FIFO que permite enviar esa estadística a un *Scope* que genera la gráfica antes mencionada.

Finalmente, para registrar el inicio de reproducción se invoca a una función cada vez que se inicia la reproducción de un segmento (a la entrada del servidor *playing* que simula la reproducción de los segmentos), dicha función registra el tiempo actual redirigiéndolo a un *Scope*. Para registrar la representación de cada segmento solicitado se invoca a otra función cada vez que se solicita un segmento (a la salida del servidor *Adaptive Algorithm*), dicha función recibe el tamaño de la representación seleccionada y lo redirige a un *Scope*. Estas métricas nos permiten observar los tiempos de interrupción de cada cliente y las fluctuaciones en la calidad del vídeo durante su reproducción.

4.2. Validación funcional

Realizamos una serie de experimentos para verificar que tanto el cliente como el servidor tenían el comportamiento esperado en distintas situaciones.

En la figura 10 detallamos los parámetros utilizados en algunos de los experimentos realizados, en cada caso se muestran los parámetros relevantes para el experimento.

Nombre	Exp1	Exp2	Exp3	Exp4	Exp5	Exp6
clients	1	2	2	2	10	10
playback_buffer_capacity	15	15	15	15	15	15
duration	2	2	2	1	1	1
threshold	10	10	10	10	10	10
delay_threshold	10	10	10	10	10	10
representations	1	1	1	1	10	1
representation_default	1	1	1	1	5	1
representation_sizes	[30]	[10]	[10]	[10]	[1,2,3,4,5,6,7,8,9,10]	[200]
simultaneousConnections	>1	1	>2	>2	>2	>10
RTT	0.01	0.5	0.01	0.01	0.01	0.01
httpQueueCapacity	15	15	1	10	10	25
httpThreads	20	20	20	20	20	10
fetch	200	200	200	200	200	150
ioBuffers	45	45	45	20	20	2
buffer_capacity	20	20	10	10	10	25
blockSize	5	5	5	5	1	5
drainTime	0.01	0.01	0.01	0.01	0.01	0.05

Figura 10: Parámetros de configuración utilizados en los experimentos.

A continuación se detallan el propósito y los experimentos realizados para validar el comportamiento del servidor.

- Experimento 1: Validar tiempo de respuesta y flujo de una entidad en el servidor:
  El propósito de este experimento era comprobar que el tiempo de respuesta del servidor era el correcto, recibiendo solicitudes de un solo cliente con recursos suficientes para satisfacerlas sin incurrir en tiempos de espera, es decir, teniendo en cuenta el tiempo de la conexión TCP, el fetch y el drenado de los segmentos solicitados. El resultado esperado para el primer segmento enviado por el cliente es: responseTime = TCP + fetch + drain = 0.01 + 30/200 + (30/5)*0.01 = 0.22 segundos, mientras que para el segundo segmento es 0.21 segundos porque no incluye el tiempo de establecimiento de conexión TCP. Obtuvimos los tiempos de respuesta esperados. Este experimento además nos permitió verificar que el servidor diferencia de forma adecuada la primer solicitud del cliente de las restantes, y que realiza de forma adecuada el drenado de un I/O buffer, es decir, pasa por el recurso de drenado la cantidad correcta de veces, y libera los recursos cuando debe liberarlos.

- Experimentos 2 y 3: Validar tiempos de espera y rechazo de conexión (conexiones simultáneas, HTTP Listen Queue).
  El propósito de estos experimentos era comprobar que el servidor rechaza las conexiones de los clientes si se alcanza el máximo de conexiones que pueden estar siendo establecidas en simultáneo o si se excede la capacidad de la HTTP Listen Queue. En el experiemento 2 se sitúa el cuello de botella en el establecimiento de conexión, al ejecutarlo se descartó la solicitud de un cliente en la etapa de procesamiento TCP. En el experimento 3 se sitúa el cuello de botella en la HTTP Listen Queue, al ejecutarlo se descartó la solicitud de 1 cliente en la etapa de procesamiento HTTP. Obteniendo en ambos casos el resultado esperado.

- Experimento 4: Validar proceso de drenado de más de un buffer.
  El propósito de este experimento era comprobar que el servidor drena correctamente los buffers de entrada/salida cuando hay más de un buffer esperando ser drenado. El experimento cuenta con 2 clientes, que solicitarán segmentos de tamaño 10 unidades, y el servidor cuenta con buffers de capacidad 10 unidades y drena de a bloques de archivo de tamaño 5 unidades, por lo que cada segmento deberá pasar 2 veces por el recurso de drenado. La cantidad de establecimientos de conexión simultáneas y la cantidad de hilos HTTP permiten que las dos solicitudes realicen las etapas TCP y HTTP en simultáneo, invirtiendo en ellas 0.06 segundos en total, y luego entren a la etapa de procesamiento I/O al mismo tiempo. En esta etapa el cliente que llegue primero demorará 0.03 segundos en completar el drenado (drena, espera, drena y se va), mientras que el cliente que llegue segundo demorará 0.04 segundos en completar el drenado (espera, drena, espera, drena y se va). Obteniendo

entonces tiempos de respuesta de 0.09 y 0.1 segundos respectivamente. Este fue efectivamente el comportamiento observado.

A continuación se realizó otro experimento (Experimento 5) para comprobar el correcto funcionamiento del cliente a través de la observación de las métricas tomadas del mismo.
En primer lugar, se observó que la carga del buffer aumentaba constantemente durante el primer segundo de simulación, hasta llegar al *delay_threshold*. Desde ese momento en adelante, la carga del buffer se mantiene en un entorno de uno del *delay_threshold*. Esto corresponde a que a partir de que la carga del buffer llegue al *delay_threshold*, se va a esperar al menos a que termine de reproducirse el segmento actual (y por ende que la carga del buffer disminuya en uno) antes de solicitar un nuevo segmento.
Luego, observando el tiempo de respuesta promedio, concluimos que el primer segmento solicitado es el que tiene mayor tiempo de respuesta (los diez clientes mandan una solicitud en el mismo instante, y todos deben realizar la conexión TCP). Luego el tiempo de respuesta promedio continúa disminuyendo hasta el segundo dos de ejecución. Esto se atribuye al desfase de las solicitudes, la disminución de los tamaños de representación, y el no tener que volver a realizar la conexión TCP. El tiempo de respuesta se mantiene relativamente estable hasta cerca del final de la ejecución, en el que aumenta ya que se comienzan a pedir segmentos de mayor tamaño, debido a la carga de los buffers.
En cuanto a los tiempos de reproducción, desde que llega el primer segmento, no se observan instancias de "buffering", en las que no se tenga nada para reproducir. Esto se pudo haber predicho simplemente observando los valores de la carga del buffer durante el tiempo de ejecución, ya que nunca se queda sin segmentos.
Por último, los tamaños de las representaciones solicitadas respaldan el resto de los datos observados en el experimento. Se comienza con una representación de calidad intermedia, que va a disminuir al comienzo dado que el buffer de reproducción comienza vacío. Una vez que lleguen suficientes segmentos al buffer como para que la carga del mismo supere la primera franja del algoritmo adaptativo, las representaciones se mantienen constantes, hasta que la carga supere la segunda franja, y es a partir de ese momento que las representaciones pedidas aumentarán su calidad, hasta llegar a la calidad máxima. Esta se mantiene por el resto de la simulación ya que la carga del buffer de reproducción nunca baja de la última franja.
En resumen, este experimento nos permitió comprobar que:

- El algoritmo adaptativo funciona correctamente: Se introduce inter-request delay y se cambia la representación cuando corresponde.
- Se piden segmentos cada vez que llega una respuesta.

Finalmente, se realizó un último experimento (Experimento 6). El objetivo de este experimento era analizar el comportamiento tanto de los clientes como del servidor en un estado de sobrecarga del servidor. Para lograr simular dicho estado se utilizó la cantidad máxima de clientes (10), se pidieron segmentos de gran tamaño (200), y se limitaron los recursos del servidor: 2 buffers de entrada/salida, fetch = 150, drain = 0.05, de manera tal que el fetch de los segmentos lleve aproximadamente 1.33 segundos y el drenado de un segmento (una vez que le es asignado un buffer) llevaría entre 3.95 y 4 segundos teniendo en cuenta que hay 2 buffers compartiendo el recurso de drenado simultáneamente. En esta situación pudimos observar, en primera instancia, que las solicitudes de los clientes eran atendidas de a pares, ya que se contaba con 2 buffers de entrada/salida, y además, se respetaba el orden de llegada de las solicitudes en la asignación de recursos. Además, observamos que se da un efecto cascada en la recepción de respuestas en los clientes, es decir, los momentos en los que reciben los segmentos se van desplazando para cada par de clientes. Este comportamiento lo podemos observar en las gráficas de la figura 11, donde mostramos como varía la carga de los buffers de reproducción de los clientes 1, 2, 5 y 6 en función del tiempo. Estas gráficas además nos permiten observar que se dan interrupciones en la reproducción de los vídeos ya que el buffer de reproducción permanece vacío por más de 1 segundo, que es la duración de los segmentos. Este comportamiento nos permitió además verificar que se piden segmentos cuando se vacía el buffer de reproducción, es por esto que los segmentos 2 y 3 llegan más cercanos en el tiempo, ya que al llegar

la respuesta del segmento 1, se pide el segmento 2, y 1 segundo más tarde al terminar de reproducir el segmento 2 se solicita el segmento 3. Finalmente pudimos confirmar que se generó una sobrecarga en el servidor ya que el mismo tuvo que descartar solicitudes, obteniendo una tasa de error mayor a 0.

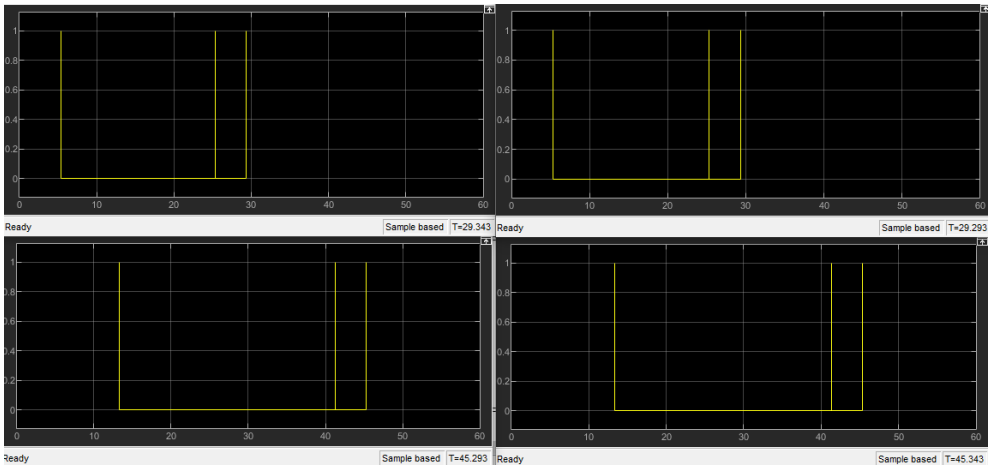

Figura 11: Carga en buffers de reproducción en función del tiempo para clientes 1 (arriba-izquierda), 2 (arriba-derecha), 5 (abajo-izquierda) y 6 (abajo-derecha).

4.3. Plan de validación

Además de la validación funcional, se debe realizar una validación enfocada a evaluar la precisión del simulador a la hora de predecir el desempeño del sistema simulado. A continuación se describe un plan de validación del simulador, aunque la ejecución del plan no está dentro del alcance del trabajo.

Como ya validamos que el cliente se comporta de la manera esperada y no contamos con un punto de comparación para el sistema completo (sistema HAS), proponemos validar el simulador del servidor web de forma independiente. Ya que de esta forma podemos controlar la tasa de arribo de solicitudes, permitiendo replicar los experimentos presentados por otros autores que hayan evaluado un simulador web de estas características.

Para que los resultados puedan ser comparables se debe ejecutar el simulador con los mismos parámetros de configuración y tomar las mismas métricas de evaluación. En su estudio, Van Der Mei et al. [2] presentan, además del modelo, los resultados de un experimento realizado para la validación de su simulador, el cual compararon contra mediciones tomadas en un laboratorio de pruebas. Por lo que consideramos un buen punto de partida comparar el desempeño de nuestro simulador contra los resultados allí presentados. Sin embargo, existe el problema de que al detallar los parámetros de configuración que fueron utilizados en el experimento existen algunas ambigüedades, en particular, no se especifica la unidad de la capacidad de los buffer de entrada salida utilizados, y para el tiempo de fetch se brinda un valor en *ms* en lugar de una velocidad, por lo que no se especifica cuántas unidades se pueden fetchear en ese tiempo. Estas particularidades hacen que el experimento no pueda replicarse con exactitud. En la figura 12 detallamos los parámetros documentados para dicho experimento, y los parámetros de configuración de nuestro simulador con los que se corresponden, marcando en rojo los valores que presentan ambigüedad.

Parámetros punto de comparación	Parámetros nuestros	Experimento 1
BIO	buffers_capacity	1024
rfetch	fetch	9.4 ms
NTCP	simultaneous Connections	1024
RTT	RTT	1 ms
NIO	ioBuffers	60
NHTTP	httpThreads	40
BHTTP	httpQueueCapacity	500
rdrain	drainTime	0.00256s
MSS	blockSize	1460 bytes

Figura 12: Parámetros de configuración del experimento presentado por Van Der Mei et al.[2]

Para poder replicar el experimento de manera fiel deberían desambiguarse los valores antes mencionados, y luego calcular la cantidad de transacciones atendidas por segundo y el tiempo de respuesta en función de la tasa de arribo de solicitudes, ya que son las métricas presentadas en el punto de comparación.

De no poder desambiguar dichos valores, debería buscarse otro punto de comparación en el que los experimentos puedan replicarse correctamente, o realizar nuestras propias mediciones en un laboratorio de pruebas contra las que compararnos.

5. Limitaciones y trabajo futuro

Si bien obtuvimos una simulación funcional del modelo diseñado, a continuación detallamos algunos aspectos que creemos pueden ser mejorados o expandidos.

Por un lado, una característica de la simulación que se diferencia del sistema real es que no modelamos la eventual finalización del vídeo, sino que el cliente seguirá pidiendo segmentos del mismo hasta que finalice la ejecución. Ya que nuestra simulación asume un único vídeo, no sería difícil mantener la duración total de los segmentos solicitados por el cliente, y no dejarlo seguir solicitando en caso de que la duración total llegue a la duración del vídeo.

Además, en la implementación actual no impactamos del lado del cliente los cierres de conexión que se dan en el servidor. Podría darse el caso en que el cliente siga mandando solicitudes a pesar de haberse cerrado la conexión, sin establecer una nueva conexión. Esto podría arreglarse haciendo que en un cierre de conexión en lugar de descartar la entidad esta sea marcada y enviada al cliente, cuando el cliente la recibe, vuelve a enviar una entidad de inicio de conexión antes de seguir enviando solicitudes de segmentos. De todas formas esto no evitaría el problema en caso de que haya más de una solicitud del mismo cliente siendo procesada por el servidor al mismo tiempo y que la primera de ellas genere un cierre de conexión, ya que la segunda se continuaría procesando de forma independiente a este suceso.

Como se explicó previamente el modelo original posee múltiples servidores, mientras que la implementación llevada a cabo solo consta de uno. Por lo tanto, a futuro sería bueno incluir un número mayor de servidores que puedan atender las solicitudes de los clientes, agregando toda la lógica requerida para esto incluyendo un balanceador de carga que se encargue de gestionar la comunicación entre clientes y servidores. Además, se podría aumentar la oferta de cada servidor, permitiendo que estos provean más de un vídeo y cada cliente seleccione el vídeo a descargar.

De la misma manera, sería beneficioso poder tener una mayor capacidad de clientes. Actualmente la cantidad está limitada por la cantidad de bloques que permite la herramienta de simulación. Una de las posibles implementaciones que discutimos consiste en que un bloque de cliente corresponda lógicamente a varios clientes, pero es una idea que aún no fue concretada.

Una vez se cuente tanto con clientes como servidores, podría implementarse la capacidad de que el balanceador de carga pueda controlar cuántos usa en tiempo de ejecución. Esto permitiría generar cambios en la carga (variando la cantidad de clientes) o reaccionar a dichos cambios (variando la cantidad de servidores).

Por último, consideramos que como trabajo futuro sería adecuado ejecutar un plan de validación para evaluar la fidelidad de la simulación. Además, en caso de utilizarse para entrenar un algoritmo de aprendizaje automático, sería conveniente registrar las métricas en archivos de configuración para poder procesarlas y analizarlas fácilmente y también realizar un análisis exhaustivo de la eficiencia (en términos de tiempo de ejecución) del simulador desarrollado.

6. Conclusiones

Concluimos que logramos obtener un diseño completo de un sistema HAS, tanto del cliente y del servidor, como de la interacción entre ellos. Además, obtuvimos una implementación funcional tanto de un cliente como de un servidor del sistema, e implementamos correctamente la interacción entre múltiples clientes y un servidor. Por otro lado, a partir de los experimentos realizados podemos concluir que obtuvimos resultados satisfactorios en cuanto al comportamiento del sistema tanto en un estado estable como sobrecargado.

Finalmente, logramos desarrollar un simulador que simula de manera fiel el comportamiento del sistema diseñado, y que posee una gran cantidad de parámetros configurables que permiten representar y modelar sistemas de diversas características.

7. Bibliografía

[1] Harald Ott, Konstantin Miller y Adam Wolisz. “Simulation framework for HTTP-based adaptive streaming applications”. En: *Proceedings of the Workshop on ns-3*. 2017, págs. 95-102.

[2] Robert D Van der Mei, Rema Hariharan y PK Reeser. “Web server performance modeling”. En: *Telecommunication Systems* 16.3 (2001), págs. 361-378.