Universidad de Buenos Aires
Facultad De Ingeniería
Año 2023 - 2do Cuatrimestre

Introducción a los sistemas distribuidos (75.43/75.33/95.60)

FECHA:2/10/2023

INTEGRANTES:

Emanuel, Tomas <temanuel@fi.uba.ar>	#108026
Gazzola, Franco <togonzalez@fi.uba.ar>	#105103
Gonzalez, Tomas <togonzalez@fi.uba.ar>	#108193
Harriet, Eliana <eharriet@fi.uba.ar>	#107205
Ramos, Federico Andrés <faramos@fi.uba.ar>	#101640
Valsagna Indri, Federico Martin <fvalsagna@fi.uba.ar>	#106010

Índice

1. Introducción

El presente trabajo tiene como objetivo la implementación de transferencia de datos confiable utilizando sockets UDP en la capa de transporte subyacente. Para esto se desarrollaron dos protocolos, *Stop & Wait* y *Selective Repeat*, que dada la naturaleza de UDP al momento de implementar los protocolos debimos considerar la perdida de paquetes, simulada a su vez con la herramienta *comcast*. Además, se realizo una aplicación cliente-servidor con motivo de poner en prueba el funcionamiento de los protocolos.

2. Hipótesis y suposiciones

Al trabajar sobre UDP se deben tener en cuenta lo siguiente

- Es un protocolo **sin conexión** y no garantiza la entrega de paquetes ni el orden de llegada. Por lo tanto se debe suponer que los paquetes pueden perderse o llegar fuera de orden.
- Proporciona un campo de *checksum* que permite detectar errores en los paquetes. El mismo descartara el paquete o entregara un error, por lo tanto supondremos que los paquetes entregados son íntegros.
- No garantiza un tiempo de entrega constante. Debemos asumir que los paquetes no tardaran un tiempo constante en llegar al receptor.
- Tamaño máximo de paquetes. Se debe tener en cuenta el tamaño máximo del paquete UDP.

Teniendo en cuenta lo anterior nuestra implementación deberá contar con:

- Un mecanismo para detectar paquetes perdidos y retransmitirlos de manera confiable para garantizar la entrega completa de los datos
- Un mecanismo para reordenar los paquetes en el orden correcto antes de entregarlos a la capa de aplicación si es necesario.
- El archivo podrá tener un tamaño máximo tal que la cantidad de paquetes en enviados no supere 2^32 paquetes.

3. Implementación

3.1. Capa de aplicación Cliente-Servidor

El servidor puede manejar conexiones de múltiples clientes al mismo tiempo. Para lograr esto, a cada cliente que se conecta al servidor se le asigna un nuevo thread, en el que se parsea el primer mensaje recibido y se detecta si el cliente quiere hacer un DOWNLOAD, UPLOAD o si no es un mensaje valido para el servidor. Una vez procesado el mensaje, se

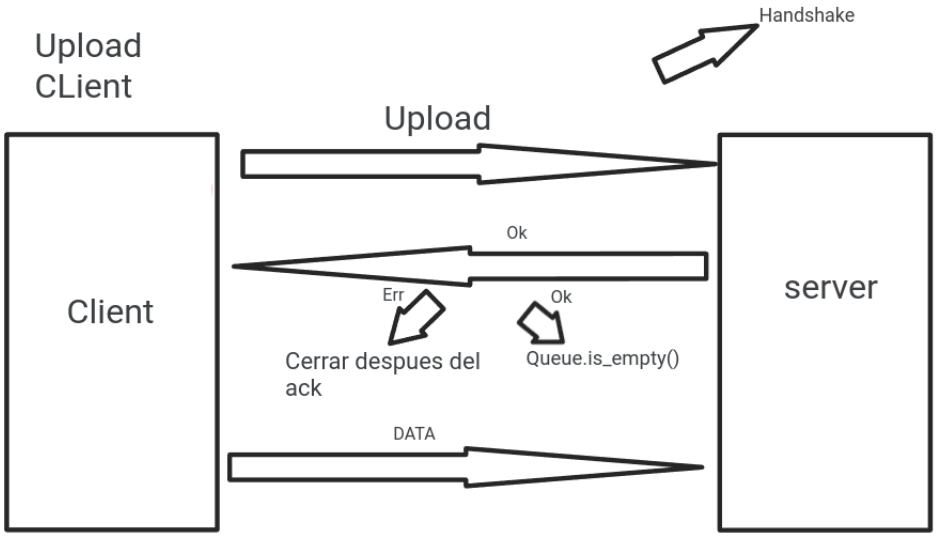
empieza a correr la funcionalidad solicitada por el cliente. En caso de haber un error en el procesamiento del mensaje, los recursos asociados a la conexión se liberan.

Debido a las limitaciones de UDP, todos los mensajes se reciben desde el mismo socket. Para cada conexión (y el respectivo cliente), se utiliza una lógica para multiplexar los mensajes a sus respectivos threads. Para esto, una vez que el servidor recibe un mensaje a través del socket, valida si la dirección de origen ya fue registrada. Si la dirección ya esta registrada, se envía el mensaje a una cola conectada al thread correspondiente. En caso de que la dirección no este registrada, se considera como un cliente nuevo y empieza todo la lógica descripta anteriormente.

3.2. Handshake

La lógica del handshake difiere para el upload y para el download dado que no se puede implementar de la misma manera ya que cada thread no posee el socket, sino que la cola de lectura.

Handshake Upload



En el Handshake del Upload del lado del cliente, al comenzar con el primer mensaje usaremos el protocolo rdt Stop and wait, por lo que el ACK se espera del lado del servidor. Si este no responde durante un Timeout especificado en *constants.py*, se le reenviará el Upload hasta que reciba el ack.

Por el lado del servidor, tenemos dos casos:

- Caso UPLOAD
- Caso Download

Cuando se recibe el paquete, el Handshake se completa. Este paquete se encola y dejamos que el protocolo se encargue de enviar el ACK para empezar con el download del lado del servidor. Pero, si recibimos un UPLOAD de nuevo, significa que se perdió el ACK, y por

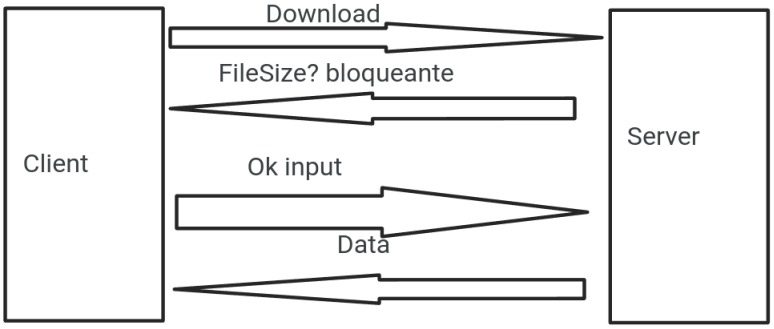
lo tanto el servidor debe reenviarlo. Este proceso se itera hasta no tener mas intentos por enviar (constante en el archivo de constants.py). Esa constante la definimos pensando en la probabilidad que se pierden los paquetes en comcast con 10 % de perdida. Al tener dos posibles paquetes por perder del lado del cliente (data y ack) y teniendo una probabilidad de 0.1 de perder un paquete, la probabilidad de que ocurra un timeout cuando se podría llegar a la conexión, dado un archivo de 5MB dividido en 1310 paquetes por enviar, será muy baja.

Si asumimos una probabilidad de pérdida de paquetes de p = 0.10, entonces la probabilidad de que un paquete se entregue correctamente es (1 - p). Si asumimos que la probabilidad de pérdida de ACKs es también p, entonces la probabilidad de que un ACK se entregue correctamente es (1 - p). Bajo este escenario, la probabilidad de que un paquete se entregue correctamente en la primera transmisión y que el ACK correspondiente también, es de $(1-p)^2$. Si, a su vez queremos calcular la probabilidad de que esto ocurra en las 10 primeras transmisiones (es decir, que no haya retransmisiones), se obtendría $(1-p)^{20}$.

Ahora, si queremos calcular la probabilidad de que haya retransmisiones, tenemos que considerar todas las combinaciones. Para cada una de estas, la probabilidad se calcula como sigue:
$P(NRetransmisionesExitosas)xP(NRetransmisionesFallidas) = (1-p)^{2n} \times p^{10-n}$

Es por ello, que teniendo un numero elevado, la probabilidad de perder una conexión cuando en realidad no se perdió es muy baja.

Handshake Download



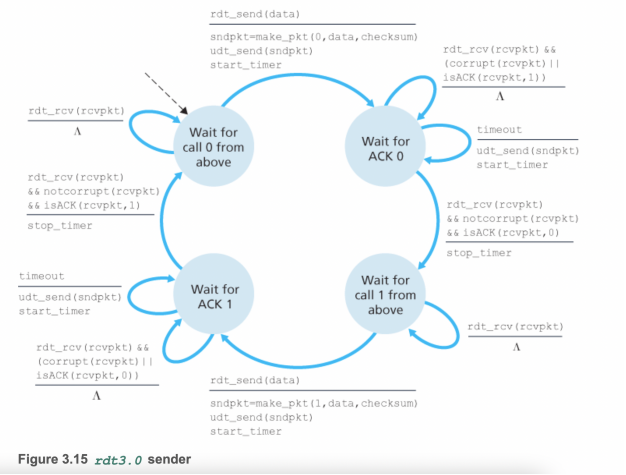
Durante el handshake del cliente, ocurre un problema, la conexión es bloqueante del lado del servidor para poder estar esperando el aceptación o no por parte del cliente, ya que este debe hacer un input si quiere o no descargar el archivo una vez que se le provee el Filesize.

Al empezar del lado del cliente el flujo es lo mismo al caso del upload, ya que estamos esperando por el filesize. Una vez que recibimos el filesize, el cliente debe esperar por la data. Si es que se pierde el filesize, saltara el timeout por parte del primer mensaje y se envia el comando del download de vuelta. Aunque este mecanismo aumenta la probabilidad calculada arriba, es el unico mecanismo capaz de esperar a que el cliente ingrese el input.
Del lado del servidor, esperamos el download, una vez que llego le enviamos el filesize, y

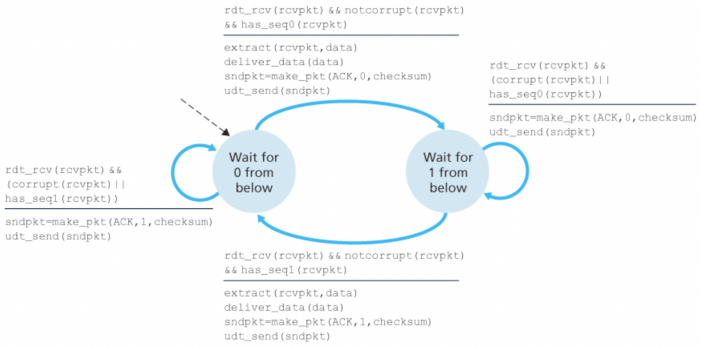
esperamos a su vez un tiempo predeterminado para saber si no se corto la conexión. Si nos llega el Download de vuelta, reenviamos el filesize, si nos llega el input, ya directamente lo direccionamos al protocolo porque en si el handshake acabo.

3.2.1. Protocolo Stop Wait

El flujo del Stop And Wait se resume en estas dos imágenes:



((a)) Stop and Wait Sender



((b)) Stop and Wait Receiver

Para ambas funciones, upload y download, se usa la misma lógica, solamente con un cambio ligero cuando recibimos las cosas.

Upload

Por el lado del Upload por el lado del cliente, al separar en paquetes, por cada paquete que enviamos esperamos recibir un ack de respuesta. Al recibir dicho ACK por el socket interno del cliente, podemos proseguir con el próximo paquete. Si salta el timeout, entonces estamos en el caso donde se haya perdido el paquete nuestro o se haya perdido el ACK del servidor (como se ve en la imagen), es por esto que reenviamos el paquete y el rdt del servidor se encarga de o agarrar el paquete o descartarlo

Mientras que del lado del servidor, como declaramos en la parte de la implementación de la aplicación, debemos multiplexar los recibidos según su address en una cola, por lo que no estaríamos recibiendo por una socket en si, sino que estaríamos recibiendo la data por la cola. Al recibir la data, ya podemos reenviar el numero de ack que se encuentra para que se encargue el sender de poder seguir con el próximo. Como se dijo, si ya procesamos el paquete, solamente lo descartamos y reenviamos el numero de ACK del paquete.

En los dos extremos, contamos con un contador que va iterando sobre 0 y 1 para poder llevar una cuenta de si avanzar o esperar como lo dicta el protocolo.

Download:

Por el otro lado, en el download usamos la misma lógica que el Upload del lado del cliente,

pero del lado del servidor, una vez mas, debemos recibir los ack que se manda del cliente una vez que recibió la data por la cola. Una vez mas, el servidor actúa como el cliente en el lado del Upload, solamente con este ligero cambio de recibir los acks por una cola. Mientras que el cliente, sigue la logica del servidor en el Upload, solamente que recibe la data por la socket en si.

Cabe decir, que definimos un Timeout lo bastante chico ya que la multiplexación y el procesamiento del protocolo es lo suficientemente como para asumir que si se transcurrió ese tiempo, se sabe que el paquete se perdió y no estar esperando un paquete que nunca llegaría, aumentando el tiempo de subida/bajada cuando se pierden los paquetes. La estructura del protocolo se encuentra abstraída en la clase RDTSWSocket.

3.3. *Selective Repeat*

La implementación del protocolo de Selective Repeat se encapsula en 3 clases principales, por un lado *SelectiveRepeatRDT* y por otro lado, dos clases subordinadas a la misma, *SenderSR* y *ReceiverSR*. Conectadas mediante 4 colas.

En cuanto a estructura, la clase *SelectiveRepeatRDT* posee dentro de si al sender y receiver, como así también el socket UDP y 3 de las colas. El *ReceiverSR* contiene a la *ReceivingWindow* donde se encuentran los paquetes recibidos a la espera de ser devueltos al usuario, estos paquetes están ordenados, en un *OrderedDict*. Finalmente, en la clase *SenderSR* se encuentra la *SlidingWindow*, el socket UDP para enviar *bytes*, y, principalmente, el sequence number, el cual lleva un conteo de los paquetes armados. En la *SlidingWindow* se encuentran aquellos paquetes ya enviados a espera de ser confirmados.

Respecto a las colas, la primera de ellas, la *data_queue*, se encarga de proveer los *bytes* leídos desde el UDP hacia el *ReceiverSR*, el cual transforma estos bytes en nuestra estructura *Packet* mediante el método *from_bytes*. Estos paquetes luego son filtrados entre aquellos que poseen data y los acknowledgement. En caso de que los paquetes posean un sequence number contenido en los limites de la ventana, se agregaran a la misma.

La segunda cola es la cola de mensajes, la misma se encarga de trasladar aquellas listas de paquetes consecutivos desde la *ReceivingWindow* hacia el *SelectiveRepeatRDT*, y por lo tanto, hacia el usuario, para su posterior procesamiento.

La tercera cola, la cola de acknowledgement, la cual conecta el *SenderSR* y el *ReceiverSR*. Cuando este ultimo recibe paquetes "ACK" los encola para que luego el sender pueda recibirlos y marcar los correspondientes paquetes dentro de la *SlidingWindow* como acknolewdged (o reconocidos).

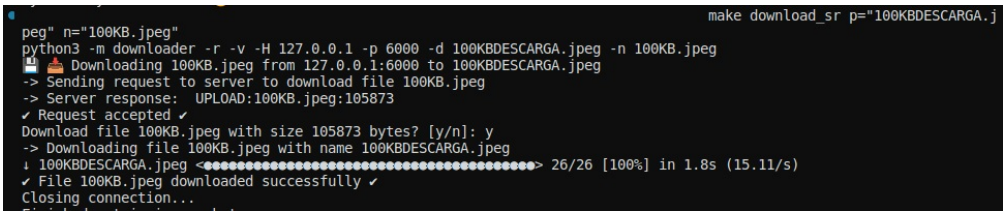
La cola de respuestas, la ultima de las colas, se ocupa de conectar la clase *SelectiveRepeatRDT* con *SenderSR*, esto puede suceder por dos motivos, el usuario recibió un mensaje previamente al cual envía una respuesta (de aquí su nombre) o en el caso de que el usuario desee enviar algo hacia la red. Dado que el envió hacia hosts externos se encuentra exclusi-

vamente en el Sender.

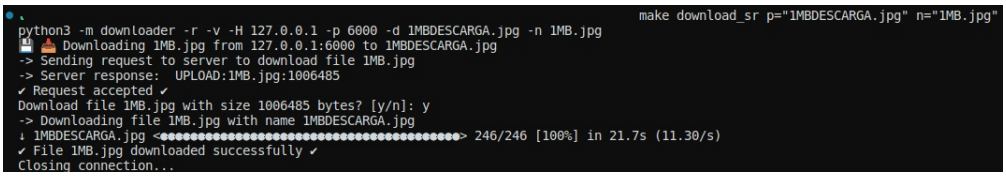
4. Pruebas

Para realizar las pruebas se utilizó la herramienta *comcast* con perdidas de paquetes de 10 %, se lo ejecutó de la siguiente manera

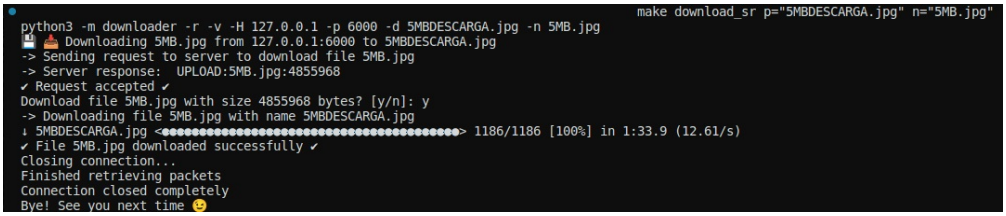
$$comcast - -device = lo0 - -packet - loss = 10\,\%$$



((a)) 100kB

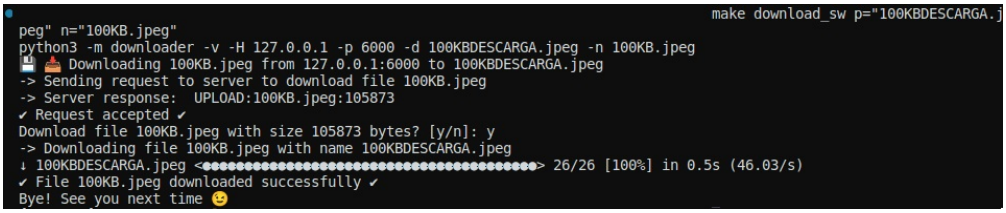


((b)) 1MB

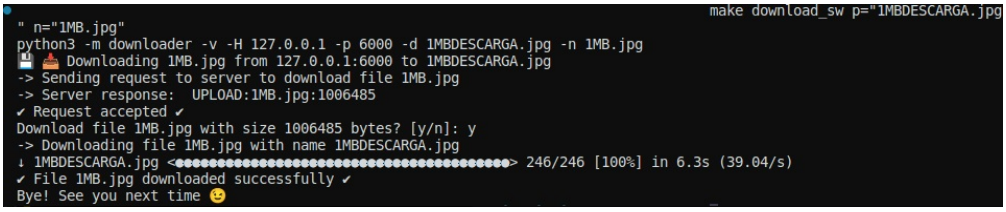


((c)) 5MB

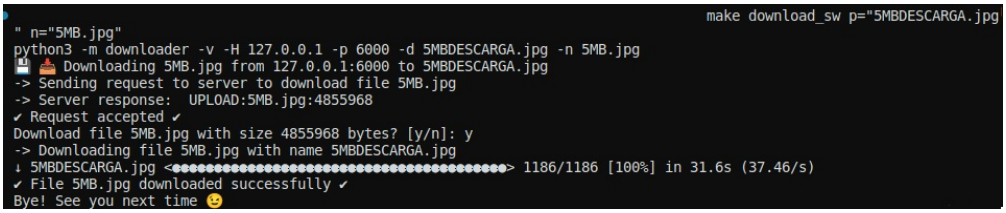
Figura 2: Download con Selective Repeat para distinto tamaños de archivo



((a)) 100kB



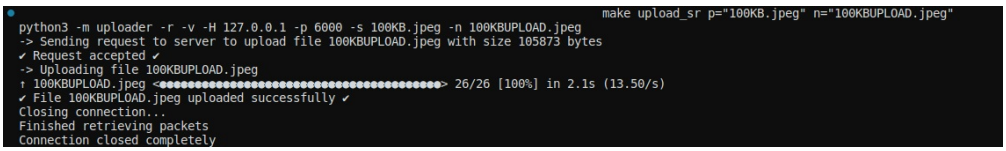
((b)) 1MB



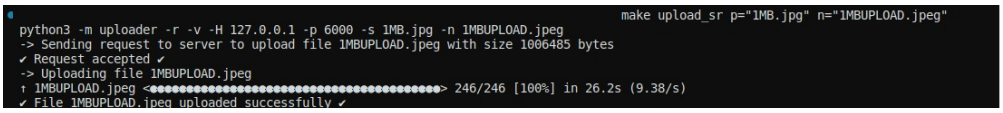
((c)) 5MB

Figura 3: Download con Stop & Wait para distinto tamaños de archivo

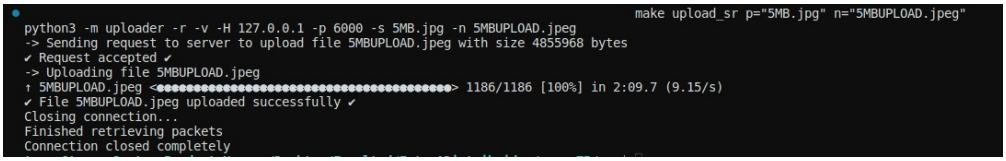
Para el download notamos que el tiempo en *Selective repeat* tiende a ser entre 2 o 3 veces mayor que en *Stop & Wait*, esto se debe a la complejidad del primero, siendo que debemos manejar arreglos de paquetes. Consideramos que al probarse en localhost, y por lo tanto, sin latencia, la recepción y envió de mensajes es instantáneo por lo que *Stop & Wait* desempeña mejor.



((a)) 100kB



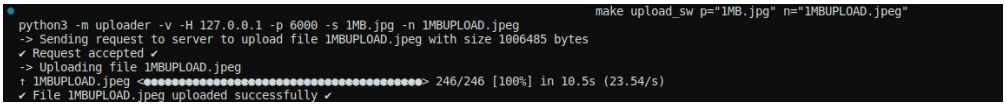
((b)) 1MB



((c)) 5MB

Figura 4: Upload con Selective Repeat para distinto tamaños de archivo

((a)) 100kB



((b)) 1MB

((c)) 5MB

Figura 5: Upload con Stop & Wait para distinto tamaños de archivo

Al momento de realizar un upload nos encontramos en una situación similar, siendo que el factor determinante para la optimización del protocolo de selective repeat es el tamaño de la ventana.

5. Constantes

En esta sección, justificamos la elección de las constantes dentro de nuestro TP:

- HARDCODED TIMEOUT

  Dicha constante es utilizada por el protocolo Selective Repeat. Hicimos pruebas para determinar cuanto tardaba el protocolo en procesar los paquetes, y en base a eso definimos un numero que no se quedaba ni muy chico para que salte el tiemout cada vez que se procesaba, ni muy grande para que el protocolo no espere demasiado una vez que se perdió.

- HARDCODED TIMEOUT SW

  Es la constante que utiliza Stop And Wait. Como el Tiempo de procesamiento de dicho protocolo es muy bajo dado que no estamos siendo afectados por la latencia, decidimos poner el numero para determinar que un paquete se perdió a su vez muy bajo.

- HARDCODED TIMEOUT FOR RECEIVING INPUT Constante utilizada para medir el tiempo en Stop and Wait en el que el protocolo espera al usuario a responder a la petición del download. Pensamos un numero que le de tiempo al usuario para poder poner una opción, ni tampoco podría ser muy grande porque el servidor estaría escuchando una conexión que se perdió sea el caso en el cual el cliente se desconecta antes de dar una respuesta.

- HARCODED BUFFER SIZE FOR FILE

Utilizamos esta constante para no tener que cargar todo el archivo en memoria cuando se trata de archivos muy grandes, definimos un numero bastante grande para poder leer de a grandes rasgos.

- HARDCODED MAX TIMEOUT TRIES

  Dicha constante ya fue descripta en la explicación del handshake durante el protocolo Stop and Wait

- PACKET HEADER SIZE

  Dicha constante hace referencia al Header que se utiliza en Stop And Wait. Al ser solamente 1 y 0 lo que recibimos en los ACKS, no necesitamos mas casilleros.

- PACKET HEADER SIZE SR

  Dado que en el header sólo se almacena información respectiva al ack almacenada en un int, el tamaño es 4 bytes.

- HARDCODED CHUNK SIZE

  Es la constante por el cual definimos la longitud de los paquetes a enviar por nuestro protocolo, es un balance entre la cantidad de paquetes a enviar y el tamaño de cada uno

  - Por el lado del Stop and Wait, a un numero mas alto, se enviarían menos paquetes y hay menos riesgo de que se pierdan. Pero a la vez, estarías cargando cada paquete con mas data.
  - Por el lado de Selective Repeat, si son tamaños grandes vas a tener menos acks, pero las ventanas van a estar más cargadas de data. Si son más chicos vas a tener windows más livianas pero muchos acks, permitiendo que los paquetes se de sincronicen más entre sí.

  Es por ello que a base de pruebas de optimización logramos encontrar un numero que se adecue a nuestras necesidades.

- HARDCODED BUFFER SIZE

  Es una Constante del resultado entre la suma del $HardcodedChunkSize$ y el $PacketHeaderSize$, para la comunicación entre los sockets.

- UPLOAD FINISH MSG

  Se utiliza para indicar desde el receptor que se termino el proceso de subida por parte del sender.

- HARDCODED MOUNT PATH

  Es el path por el cual el servidor va a guardar los archivos que se le envia desde el sender.

- WINDOW SIZE Es el tamaño de la ventana de envío, es la máxima cantidad de paquetes que aceptamos tener sin el ack correspondiente. Se eligió esta cantidad para poder enviar en simultáneo una buena cantidad de paquetes, pero que no sea demasiado grande de manera que, al momento de tener un timeout, no haya que reenviar una cantidad demasiado grande de paquetes

6. Preguntas a responder

6.1. Arquitectura Cliente-Servidor

Dentro de la arquitectura cliente-servidor existen 2 tipos de end-hosts. El servidor es un host que está siempre activo cuyo objetivo es responder las consultas que le hacen el otro tipo de hosts, los clientes. Los clientes se encargan de establecer la comunicación con el servidor. Una vez establecida, el flujo de comunicación es de consultas de parte del cliente y respuestas del lado del servidor. Bajo esta arquitectura nunca un cliente se comunica directamente con otro cliente, siempre en esos casos se encuentra el servidor como intermediario. Por ejemplo al enviar un mail no es que el mail va del usuario A al B directamente, sino que el usuario A envia el mail al servidor y luego el usuario B lo obtiene del servidor. Para garantizar la comunicación se emplea un protocolo de capa de aplicación tal que los mensajes puedan ser elaborados y comprendidos por ambas partes.

6.2. ¿Cuál es la función de un protocolo de capa de aplicación?

La función de la capa de aplicación es establecer el set de reglas con las cuales se comunican las aplicaciones en diferentes dispositivos. Siempre que el cliente construya el mensaje y se comunique como lo establece el protocolo, este garantiza que el servidor logre entenderlo.

Si bien los errores producto de las capas inferiores repercuten en la capa de aplicación, como puede ser timeout o invalid ip, la capa de aplicación se encarga de manejar aquellos errores inherentes a la misma. Por ejemplo, cuando el cliente quiere descargar un archivo que en el servidor no existe, o subir un archivo con un nombre ya existente. Si deseamos extender estos ejemplos a protocolos ya existentes, podemos mencionar como HTTP comunica a un cliente que la URL a la que quiere ingresar no existe.

Continuando con HTTP, para crear una página web es necesario aplicar este protocolo, y para visitar esa página es necesario hacerlo mediante un navegador que sepa leer ese protocolo, dado que en caso de no adherirse a estas reglas impuestas la comunicación será imposible y no se llegará a un resultado favorable.

6.3. Nuestro protocolo de aplicación

6.4. La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP

¿Qué servicios proveen dichos protocolos? ¿Cuáles son sus características? ¿Cuando es apropiado utilizar cada uno? UDP solo provee 2 servicios, por un lado el multiplexado/de-

multiplexado. El objetivo del multiplexado es garantizar que un paquete que llegó al host sepa exactamente a que socket debe ir mediante el uso de puertos que se proveen en el header del paquete. Mientras que el objetivo del multiplexado es agregar el header correspondiente a los datos a enviar y entregarselo a la capa de red. Por otro lado, provee el servicio *checksum*, el cual busca determinar la integridad de la información de un paquete. Sin embargo en la práctica solo funciona si se corrompió únicamente un bit del paquete, puede suceder que si fallan 2 bits el checksum no detecte estos errores. Por este punto es que si bien provee este servicio no termina siendo suficiente.

TCP provee a su vez servicios adicionales, dado que también multiplexa/demultiplexa y realiza checksum. El servicio más importante que provee TCP es la garantía que al enviar un paquete, el destino lo recibe correctamente, dado que en caso contrario ocurre una excepción que termina todo tipo de conexión. A su vez, asegura que los paquetes lleguen en el orden correspondiente de cara a la aplicación. Los otros servicios adicionales que provee TCP son el control de congestión y el control de flujo.

El objetivo del control de flujo es que el emisor no sobrecargue el buffer del receptor, esto se logra mediante el campo rwnd del header de TCP donde el receptor le avisa al emisor el tamaño disponible de su buffer, para que este no envie más paquetes de los que está dispuesto a recibir.

El objetivo del control de congestión es no sobrecargar la red de paquetes. Se usan los mecanismos de slow start, congestión avoidance y fast recovery para achicar las ventanas de envío cuando hay pérdidas de paquetes o timeouts.

Una característica de TCP es que es establece una conexión una vez concretado el handshake, en donde puede haber un ida y vuelta de datos entre ambos y cuando uno de los 2 cierra la conexión el otro es avisado.

TCP nos termina brindando muchos servicios adicionales, pero eso tiene el costo de un header más grande (En concreto generalmente de 20 bytes a diferencia de los 8 bytes de UDP). Esto hace que el envío de paquetes en UDP sea más liviano, lo cual termina siendo su principal ventaja.

En concreto, es conveniente utilizar TCP cuando queremos garantizar que la información que enviamos se reciba correctamente, ya sea transferencia de archivos, el cargar una página web, el envío de mensajes de texto. En todos esos casos es inadmisible que un paquete del total no llegue. UDP se utiliza cuando no es imprescindible el contenido de cada paquete, y es más importante el que lleguen los nuevos antes que reenviar los anteriores. Por ejemplo en todo lo que es streaming de video se usa UDP.

7. Dificultades encontradas

7.1. Stop And Wait

Las dificultades encontradas en el protocolo Stop and Wait se baso en que en un principio habíamos entrelazado la capa de aplicación con el protocolo Stop and Wait, esto llevo a que a la hora de probar todo, se encuentre encapsulado en un solo lugar.

Al tener abstraída la capa de aplicación de la del transporte, pudimos poder comprender bien cuando estábamos en la capa de transporte y por ende poder debuggear tranquilamente.

Los problemas encontrados durante dicho protocolo fue el hacer en si funcionar el protocolo y poder estar recibiendo exitosamente la data/ack y sincronizarlos a la vez, de no subir el contador cuando no recibiamos el paquete y la de no subir una vez que recibiamos el paquete repetido.

7.2. Selective Repeat

La mayoría de las dificultades cayeron sobre la detección de casos bordes y evaluación de la mejor pauta a llevar a cabo en dichos casos.

Un ejemplo de esto fue al momento de recibir correctamente cada paquete, enviar su correspondiente ACK, que el mismo sea recibido pero de todas formas encontrar fallas al momento de ver cómo resultó descargado un archivo. Problema que resultó de darle paquetes repetidos al usuario, la pauta tomada para esto fue no agregar paquetes ya recibidos a la ventana de recepción pero de todas formas enviar su ACK.

Otro problema fue la facilidad para ejecutar el método 'send', el problema que resulta de esto es la facilidad para sobrecargar a la ventana de envío. Si la ventana de envío está muy cargada, y se pueden seguir agregando paquetes para enviar, resulta en un desbalance que permite que la aplicación que utiliza el protocolo avance mucho más rápido de lo que el mismo avanza, resultando poco transparente el estado global de los paquetes a enviar. Para esto se optó por que el método 'send' sea bloqueante al momento de estar muy cargada la ventana, y desbloqueándose al momento en que la tasa de paquetes pendientes baja.