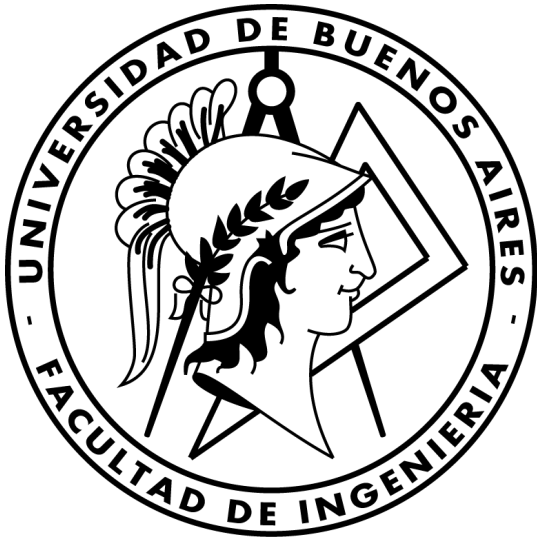


Trabajo Práctico 2

Introducción a los Sistemas Distribuidos - 75.43

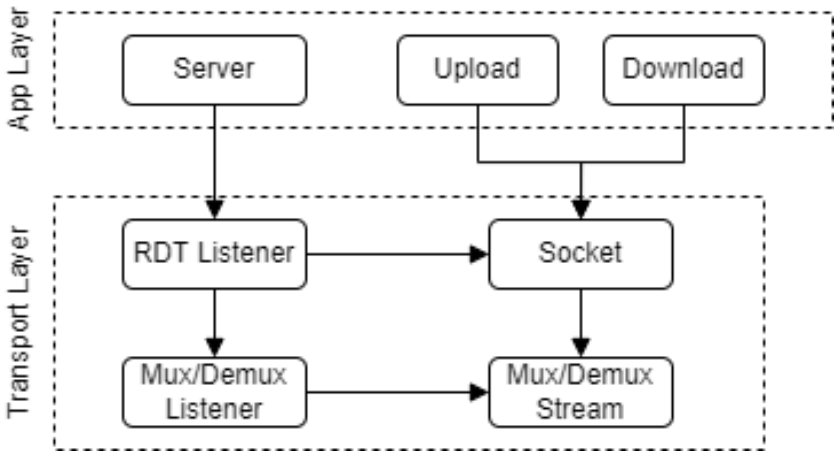
NOMBRE	PADRÓN
Armando Civini	104350
Ignacio Brusati	105586
Felipe de Luca Andrea	105646
Nicolas Zulaica Rivera	105774
Elián Foppiano	105836

Tabla de Contenido

Introducción

El presente informe detalla el desarrollo de una aplicación de transferencia de archivos, mediante el uso de un protocolo de transferencia confiable implementado sobre UDP.

Para ello se utilizó una arquitectura cliente / servidor, separando las responsabilidades de las capas del siguiente modo:



Se implementan la entrega confiable de dos maneras alternativas: Stop & Wait y Selective Repeat. Teniendo un tipo de socket para cada uno de ellos.

Hipótesis y Supuestos

- Si un cliente sube un archivo con un nombre que ya se encontraba almacenado en el servidor, entonces el archivo anterior será reemplazado por el nuevo.
- Un archivo puede ser descargado en paralelo por múltiples clientes.
- No se considera relevante el acceso a un archivo que esta siendo subido (subida simultáneas o lectura mientras se sube)

Implementación

La implementación consta de una capa de aplicación, sobre la cual se desarrolló un protocolo de transferencia de archivos tipo TLV, y una capa de transporte, que garantiza la entrega de datos confiable (RDT) a la aplicación. Con el fin de disminuir el acoplamiento del trabajo, la capa de transporte se subdividió en dos: una primera capa, que llamamos “capa Mux/Demux”, y otra llamada “capa RDT”. Las características, supuestos y garantías de cada una de las capas se detallan a continuación.

Capa de Transporte - Mux/Demux

Esta subcapa utiliza los servicios que provee UDP (multiplexación/demultiplexación de paquetes entre procesos y chequeo de errores) para proveer un mecanismo de multiplexación/demultiplexación entre distintas conexiones. El motivo detrás de esto, es que los sockets UDP se identifican únicamente con la dirección IP y puerto de destino. Por ejemplo: si dos hosts envían un paquete a 192.168.0.1:3030, ambos se recibirían por el socket que se encuentre *bindeado* a la dirección 192.168.0.1:3030 y que realizó una llamada al método *recv_from*. La forma en que se implementa esta capa es sencilla: un thread recibe paquetes UDP continuamente, y en función de la dirección de origen (que se obtiene también de *recv_from*), se encolan en diferentes *queues*. Cada una de las *queues* está asociada a un socket, y cuando éste realice la llamada a *recv*, leerá de la *queue* correspondiente.
El protocolo de esta capa cuenta con un único tipo de paquete, formado por una palabra mágica (magic word), y el payload. La palabra mágica garantiza que el paquete UDP recibido corresponde a este protocolo.

Capa de Transporte - RDT

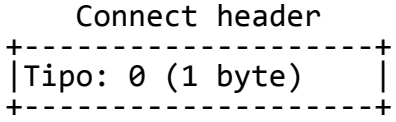
Utilizando el servicio de la subcapa anterior, provee el servicio de transferencia de datos confiable. Se implementó de dos formas diferentes: con la técnica *Stop and Wait*, y *Selective Repeat*. El protocolo es distinto en ambos casos: si bien los tipos de paquetes y headers coinciden, la forma de utilizarlos es distinta. Ambos utilizan el formato **big-endian** para la codificación de datos.

Stop and Wait

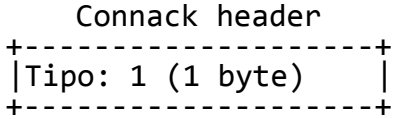
La idea básica de este protocolo es enviar un paquete, esperar un acknowledge de dicho paquete, y luego enviar el siguiente. Si se produce un timeout esperando el acknowledge, se envía de nuevo.

El protocolo consiste en 6 paquetes diferentes

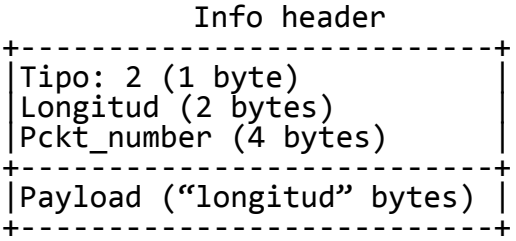
- **Connect** packet: enviado por el socket que quiere iniciar la conexión (socket de cliente). No tiene payload.



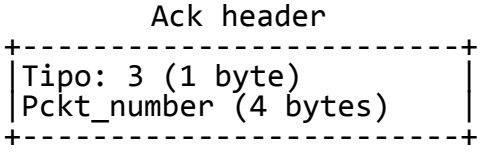
- **Connack** packet: enviado por el socket que acepta la conexión (socket de servidor) una vez que recibió el paquete connect. No tiene payload.



- **Info** packet: es el que contiene la información que se desea transportar a través del socket. Dicha información se encuentra en el payload del paquete. El número de paquete permite reconocer retransmisiones de paquetes. Por ejemplo: si un socket espera recibir el paquete número 15, y recibe el número 14, se sabe que se trata de una retransmisión. El paquete se dropea, aunque el acknowledgement se envía de todas maneras, ya que la retransmisión probablemente se deba a la pérdida del paquete Ack. La longitud del paquete debe ser menor o igual que 65514 (límite impuesto por el tamaño máximo de paquete UDP, menos los headers UDP, palabra mágica de la capa Mux/Demux y los headers del paquete Info)



- **Ack** packet: cuando un socket recibe un paquete Info con el número de paquete **x**, ya sea un paquete nuevo o una retransmisión, se envía un Ack packet con número **x**. Se requiere identificar el número de paquete en el Ack, porque una demora en la red puede provocar que este paquete se envíe por duplicado. Por ejemplo, si se envía un paquete Info y su retransmisión, ambos llegan a destino y son respondidos con un paquete Ack. Si no existiera el número de paquete en el Ack, el emisor creería que ambos corresponden a paquetes distintos. Este paquete no tiene payload.



- **Fin** packet: se envía cuando un socket quiere finalizar la conexión. Le indica al socket receptor que ya no tiene información que enviar, aunque puede seguir recibiéndola (nótese la diferencia con la implementación de *Selective Repeat*). No tiene payload.

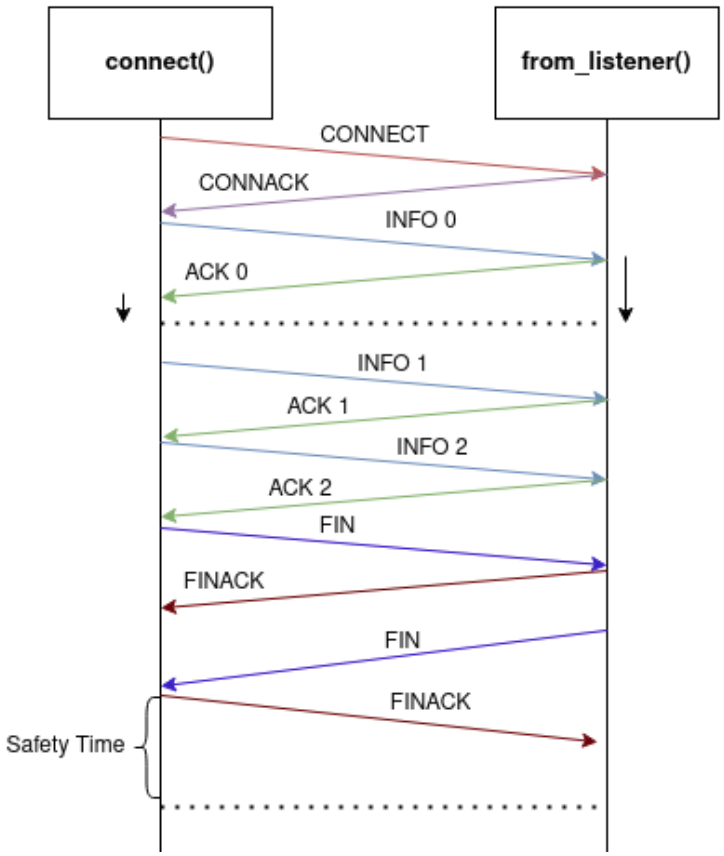
```
Fin header
+--------------------+
|Tipo: 4 (1 byte)    |
+--------------------+
```

- **Finack** packet: se envía como respuesta a un paquete Fin. No tiene payload.

```
Finack header
+--------------------+
|Tipo: 5 (1 byte)    |
+--------------------+
```

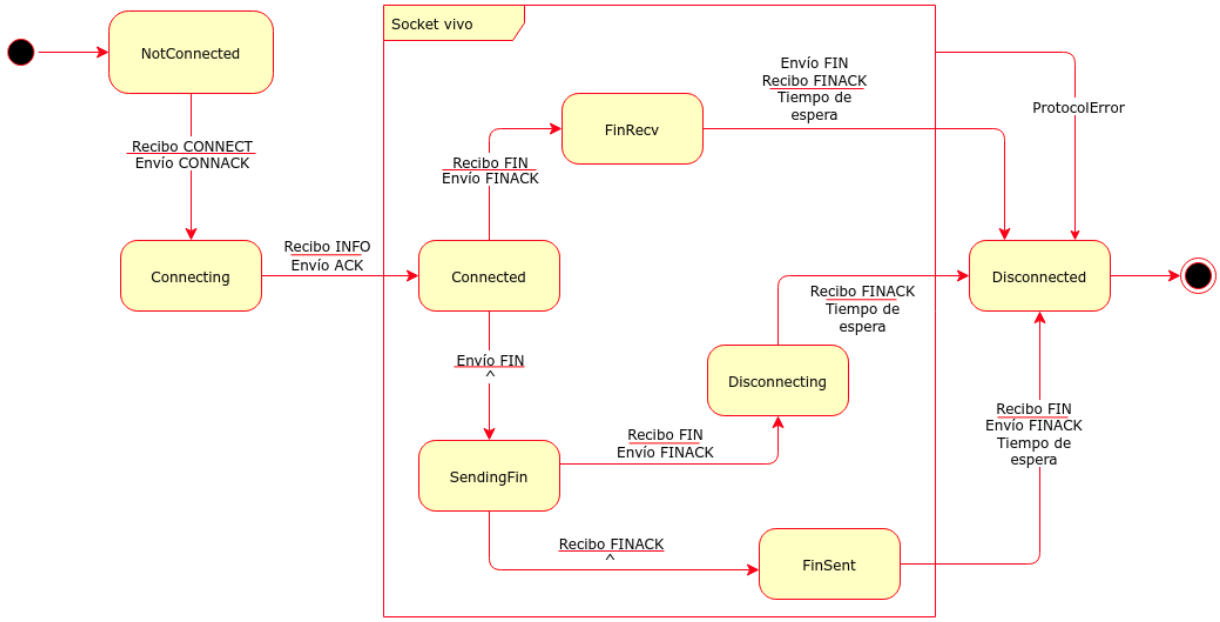
El protocolo puede ser representado mediante una máquina de estados, de forma que un socket se encuentra siempre en un estado, dependiendo de los paquetes que se enviaron y recibieron. Se presenta a continuación un diagrama simplificado de los estados y transiciones, tanto para el socket que inicia la comunicación (socket de cliente) como para el socket que la acepta (socket de servidor). Nótese que la única asimetría entre ambos diagramas es durante el inicio de la conexión.

Implementación

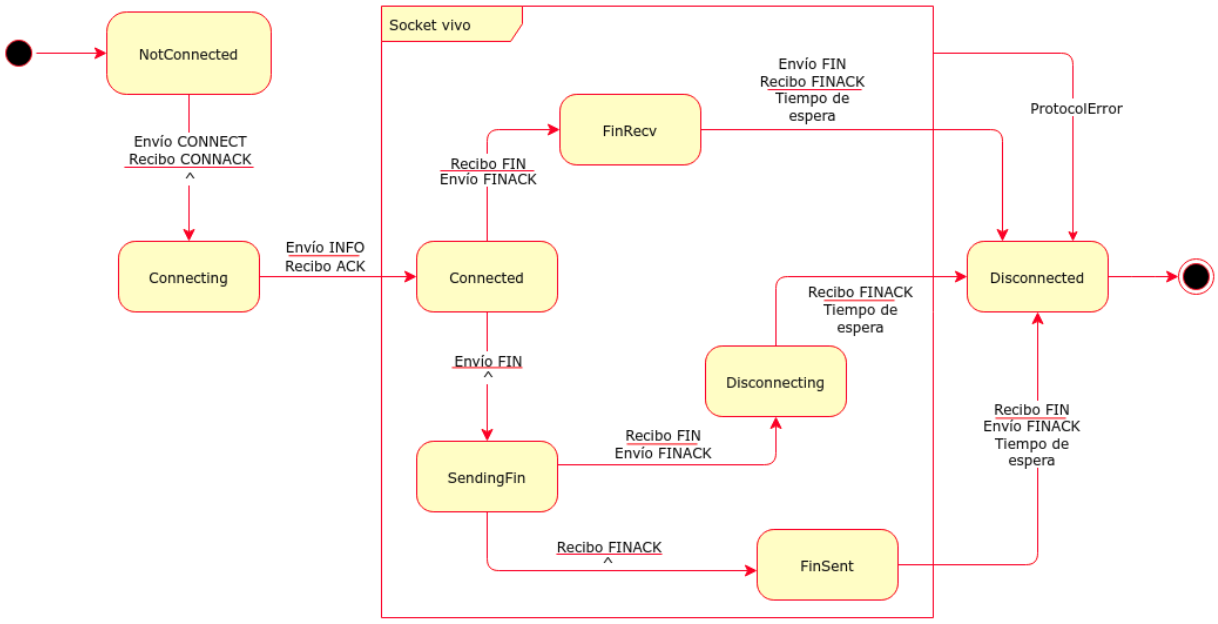


Para implementar el protocolo, se utilizó el patrón State con las siguientes transiciones:

Socket de Servidor



Socket de Cliente



Selective Repeat

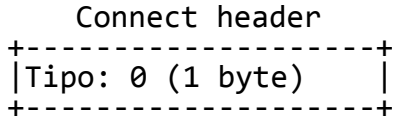
La idea básica de este protocolo es tener una **ventana de paquetes** en tránsito, a modo de poder enviar **múltiples paquetes** de manera **simultánea** y reenviar solamente aquellos paquetes que no hayan llegado al destino.

El protocolo implementado está basado en la **conexión**, y asegura la **entrega confiable** mediante el reconocimiento de los paquetes enviados.

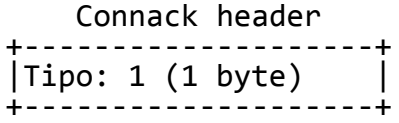
Se sabe que un paquete llegó a su destino cuando es **reconocido** (acknowledged) por la otra parte. Si un paquete no es reconocido antes de un tiempo determinado (ACK_TIMEOUT) se intenta **reenviarlo** un número determinado de veces (ACK_RETRIES) antes de dar la conexión por perdida.

El protocolo consiste en 6 tipos de **paquetes** diferentes

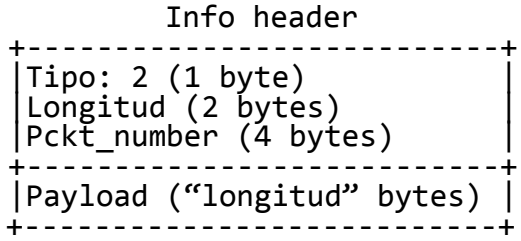
- **Connect** packet: enviado por el socket que quiere iniciar la conexión (socket de cliente). No tiene payload.



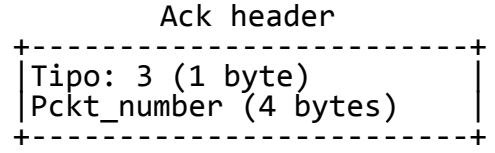
- **Connack** packet: enviado por el socket que acepta la conexión (socket de servidor) una vez que recibió el paquete connect. No tiene payload.



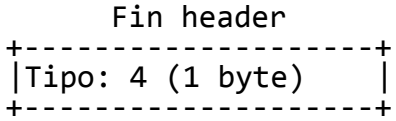
- **Info** packet: es el que contiene la información que se desea transportar a través del socket. Dicha información se encuentra en el payload del paquete. El número de paquete permite reconocer retransmisiones de segmentos y ordenarlos en el host receptor, además de identificar cual es el paquete más antiguo que no recibió el correspondiente ACK para ir moviendo la ventana acordemente. La longitud máxima es la misma que para el Stop & Wait.



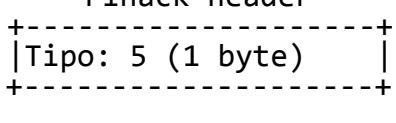
- **Ack** packet: cuando un socket recibe un paquete Info con el número de paquete **x**, ya sea un paquete nuevo o una retransmisión, se envía un Ack packet con número **x**. Se requiere identificar el número de paquete en el Ack, para mantener la secuencia de los paquetes y transmitirlos a la capa superior de manera ordenada. Este paquete no tiene payload.



- **Fin** packet: se envía cuando un socket quiere finalizar la conexión. Solo lo envía el primer host en llamar al método *close()*. No tiene payload.



- **Finack** packet: se envía como respuesta a un paquete Fin. No tiene payload.



El protocolo define los siguientes **procedimientos**

- **Conexión:**
  1. El cliente envía un paquete CONNECT;
  2. El servidor responde con un CONNACK;
  3. El cliente envía un paquete INFO con n=0, payload=None;
  4. El servidor responde con ACK con n=0;

  Si un paquete no tiene respuesta, se intentara reenviarlo luego de un tiempo determinado (timeout), una cantidad determinada de veces (retries)

- **Envío de información**
  1. El emisor envía un paquete INFO con un determinado n;
  2. El receptor envía un paquete ACK con el mismo n;

  Si un paquete no tiene respuesta, se intentará reenviar luego de un tiempo determinado (timeout), una cantidad determinada de veces (retries).
  Tanto el cliente como el servidor pueden ser emisor o receptor.

- **Fin de Conexion**

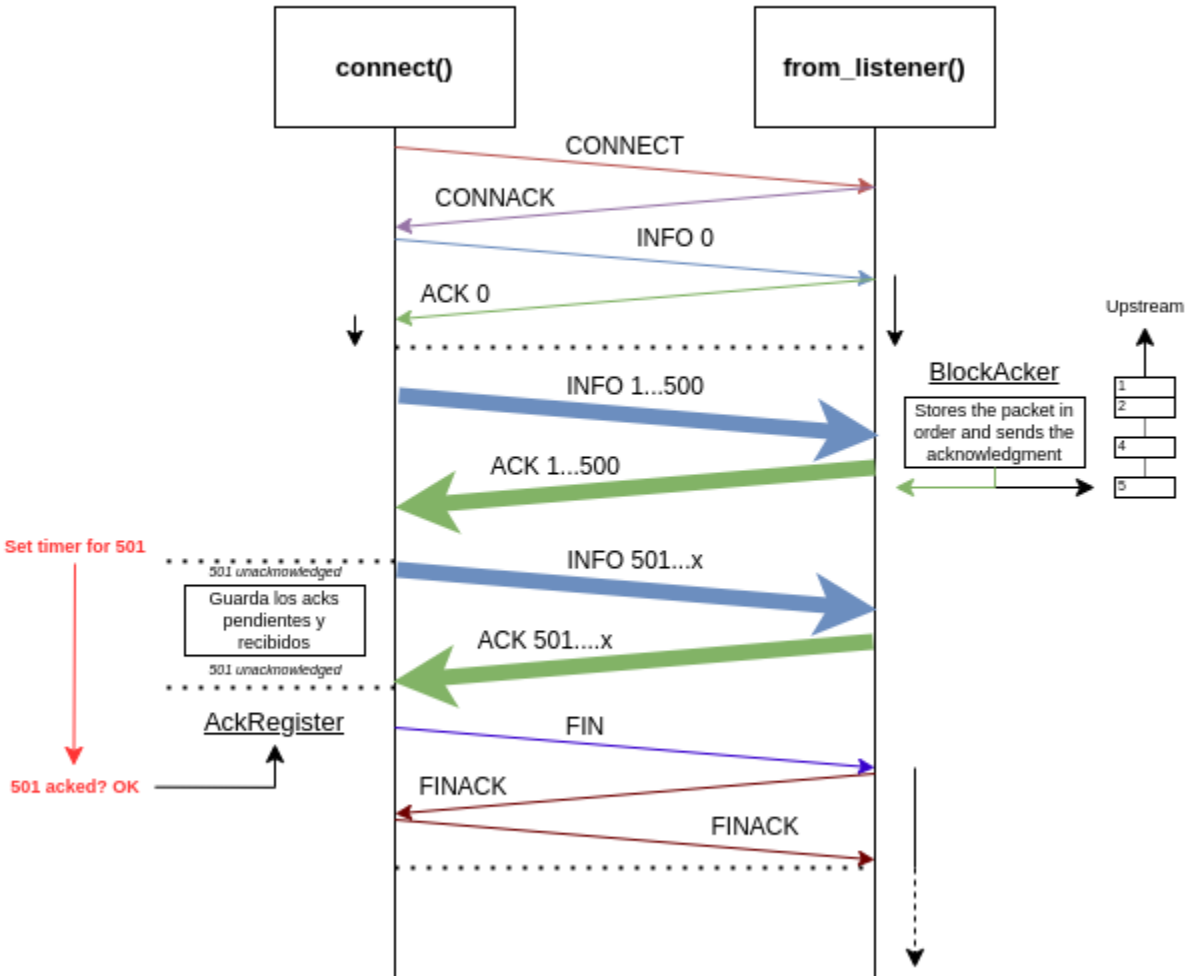
  1. El “iniciador” envía un paquete FIN;
  2. El otro responde con un paquete FINACK;
  3. El “iniciador” responde con un FINACK

  La conexión se cierra en ambos sentidos al mismo tiempo. No hay conexiones “half-open”.

  El tercer FINACK es en cierta manera “opcional”. Luego del paso 2, ese host espera recibir otro FIN en caso de que se haya perdido su FINACK hasta un timeout en el que asume que llegó correctamente, y cierra la conexión. Si llegara el FINACK del paso 3 en vez de otro FIN, ya sabe que le llegó correctamente, por lo que puede salir del bloqueo antes.

  Si ambos participantes inician el fin de la conexión simultáneamente (reciben un FIN esperando un FINNACK), ambos pasan al paso 2.

Un típico flujo con tamaño de ventana 500 en este protocolo podría ser el siguiente:



El **BlockAcker** y **AckRegister** son 2 clases definidas en el código.

El **BlockAcker** se encarga de recibir los segmentos y mandar el acknowledge. Una vez se recibe un segmento, se agrega a un buffer para enviar a la capa de aplicación cuando se tenga un bloque ordenado posterior a lo último que se envió.

El **AckRegister** guarda los Acknowledges pendientes por parte del otro host. Cuando enviamos un INFO se agrega como pendiente, y al recibir su correspondiente ACK se remueve. Luego, los timers de reenvío pueden preguntar a esta clase si se recibió el Acknowledge correspondiente.

También hay una clase **AckNumberProvider** no presente en el diagrama, que controla los números de los paquetes que se envían. Es quien garantiza que al recibir un ACK con número de paquete *n* se libere el número *n + window_size* para enviar otro INFO, y que solo se libere el número *n* si ya se liberó el *n-1*, y así desplazar la ventana.

Capa de Aplicación

La capa de aplicación se encarga de definir un protocolo para la comunicación entre el cliente y el servidor haciendo uso de la capa de transporte. Un estándar que sostiene esta capa de aplicación es que se usa **little endian**. Cada programa de la capa de aplicación recibe ciertos argumentos para llevar a cabo sus tareas, los siguientes argumentos son generales para los 3 programas:

-h , *-- help*: de incluir este, en vez de llevar a cabo el proceso se imprimirán todos los argumentos y sus descripciones.
-v , *-- verbose*: lo impreso por el programa será mucho más descriptivo.
-q , *-- quiet*: solo se imprimirán los errores (*quiet* y *verbose* no pueden estar simultáneamente como argumentos).
-H , *-- host*: este parámetro es acompañado por el número de IP del servidor.
-p , *-- port*: este parámetro es acompañado por el número de puerto donde el servidor escucha la conexión.

Argumentos específicos para upload y download:
-s , *-- src/-d* , *-- dst*: es el directorio de dónde se sacará el archivo para subir o donde se guardará el archivo para descargar.
-n , *-- name*: el nombre del archivo.

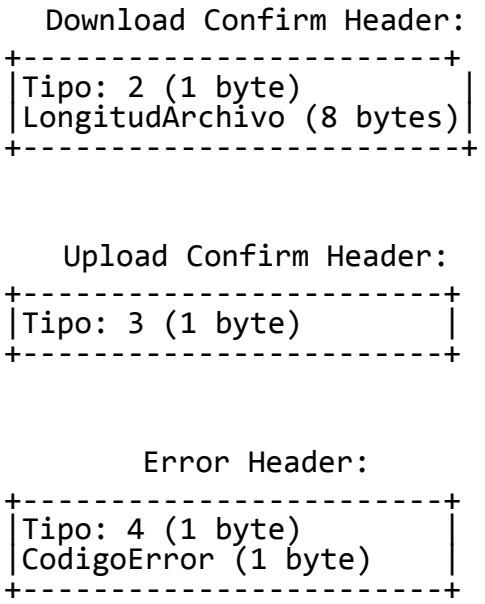
Argumentos específicos del servidor:
-s , *-- storage*: el directorio donde el servidor guardará los archivos.

A continuación se ven los distintos tipos de mensajes.

Server
El servidor cuando recibe una conexión la acepta y procede a lanzar un nuevo hilo para manejarla. Éste tiene dos funciones, recibir archivos para almacenar o enviar archivos que el cliente descarga. El servidor primero lee el primer byte del paquete para ver cual de estas dos funciones quiere realizar el cliente.

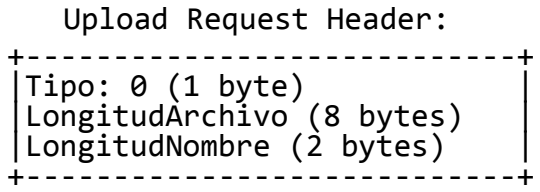
Si el primer byte es de tipo 0 (**Upload Request Header**), el servidor procede a leer 8 bytes de la conexión para saber el largo del archivo que recibirá y luego lee 2 bytes para saber el largo del nombre del archivo. A continuación se leen la cantidad de bytes que indique el largo del nombre del archivo y estos se toman como el nombre del archivo. Después el servidor leerá tantos bytes como largo del archivo que se haya recibido y eso será el contenido que escribirá en el *--storage*. Si este proceso se termina exitosamente se le enviará al cliente un paquete con nada más que el **Upload Confirm Header**. De lo contrario se responderá con un paquete que posea el **Error Header** con un código de error igual a 0 que representa cualquier error desconocido que ocurra en el servidor.

Si el primer byte es de tipo 1 (**Download Request Header**), el servidor leerá 2 bytes para saber el largo del nombre del archivo solicitado y luego leerá la cantidad de bytes indicados por éste. Con estos bytes el servidor busca el su -- *storage* si hay un archivo con el mismo nombre. De no haberlo, se enviará un paquete con el **Error Header** con un código de error igual a 1 que representa que no se encontró el archivo deseado en el servidor. De sí encontrarse el archivo, el servidor enviará un paquete de tipo 2 (**Download Header**) seguido por 8 bytes que indican el tamaño del archivo que el cliente descargará. Después de esto se enviarán los datos del archivo.



Upload

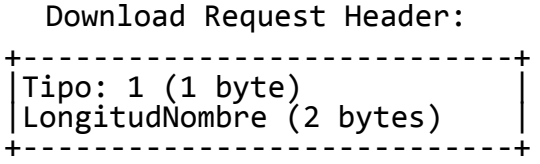
El cliente al querer subir un archivo al servidor enviará un paquete del formato **Upload Request Header** que está formado por el byte de tipo seteado en 0, 8 bytes que representan la longitud del archivo y 2 bytes que representan el largo del nombre del archivo deseado. Luego, envía el nombre del archivo y finalmente los datos del archivo. El cliente esperará una respuesta del servidor que puede ser un **Upload Confirm Header** o un **Error Header**. De recibir un **Error Header**, podrá leer del mismo el código de error y mostrarlo por pantalla.



Download

El cliente solicita un archivo para descargar mediante un paquete con el **Download Request Header**, éste comienza con el byte de tipo seteado en 1 y luego 2 bytes que representan el largo del nombre del archivo deseado. Finalmente se envía el nombre del archivo en bytes.

Después de esto el cliente espera la respuesta del servidor. De esta forma puede recibir un **Error Header** o un **DownloadConfirmHeader**. En caso de ser un **DownloadConfirmHeader** este contendrá un byte seteado en 2 y 8 bytes con la longitud del archivo. El cliente finalmente leerá esa cantidad de bytes para guardarlos como el contenido del archivo con el título enviado en -- *dst*.



Pruebas

Se verifica el tiempo de subida para diferentes configuraciones de tamaño de archivo y pérdida de paquetes. Los parámetros utilizados para la ejecución de las pruebas es:

- MSS: 128 bytes
- Tamaño de la ventana: 500 (Selective Repeat)

Stop and Wait

Tamaño (KiB)	Perdida de paquetes		
	0%	**15%**	**50%**
0.5	9.13 s	9.12 s	24.16 s
10	16.90 s	48.60 s	174.35 s
100	87.81 s	275.87 s	1180.65 s

Selective Repeat

Tamaño (KiB)	Perdida de paquetes		
	0%	**15%**	**50%**
0.5	2.01 s	3.52 s	6.62 s
10	2.11 s	5.05 s	23.06 s
100	2.34 s	12.85 s	32.15 s

Comparacion

Tabulamos en qué **factor** es mayor el tiempo que toma utilizando SAW en comparación a SR para cada configuración y su promedio.

Tamaño (KB)	Perdida de paquetes		
	0%	**15%**	**50%**
0.512	4.5	2.6	3.6
10	8.0	9.6	7.6
100	37.5	21.5	36.7
Avg.	**14.6**		

Preguntas a responder

1. Describa la arquitectura Cliente-Servidor.

En la arquitectura cliente-servidor hay un host denominado servidor que tiene una dirección IP fija y conocida y siempre se encuentra activo esperando por las peticiones de otros hosts denominados clientes. Bajo esta arquitectura, dos clientes no se comunican directamente entre sí, sino que deben hacerlo mediante el servidor. Dado que un servidor puede recibir mensajes de muchos clientes, es común que el servidor esté compuesto por varios hosts que se encuentran almacenados en un gran centro de datos. Es importante destacar que en este tipo de arquitectura el cliente debe iniciar la comunicación con el servidor y nunca puede ocurrir al revés.

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

La función de un protocolo de capa de aplicación es estandarizar la comunicación entre aplicaciones de usuario (o procesos), para ello define los siguientes elementos:

- Mensajes de petición y respuesta
- Sintaxis de los mensajes
- Campos - función, tamaño y delimitadores
- Procedimiento de envío de mensajes y sus respuestas

3. Detalle el protocolo de aplicación desarrollado en este trabajo.

En este trabajo se implementó un protocolo de file transfer. En este protocolo se encuentran un cliente y un servidor que mediante una conexión confiable se pueden enviar archivos mutuamente. El cliente puede iniciar una conexión para subir un archivo al servidor o puede iniciar una conexión para descargar un archivo al servidor. El servidor, por su parte, debe estar activo esperando por las conexiones de los clientes y debe poder almacenar los archivos que los clientes envíen y ser capaz de transferirlos en caso de que un cliente los pida. El protocolo define la forma en la que recibirá las peticiones de los clientes y la forma en la que enviará la información a los mismos. Los clientes, para hacer un uso correcto del servidor, deben estar al tanto de las convenciones utilizadas para poder enviar y recibir los datos de una forma satisfactoria.

Se pueden encontrar los tipos de paquetes y procedimiento del protocolo en la sección de implementación.

4. La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP. ¿Qué servicios proveen dichos protocolos? ¿Cuáles son sus características? ¿Cuándo es apropiado utilizar cada uno?

UDP proporciona los servicios mínimos de un protocolo de capa de transporte:

- Multiplexado / Demultiplexado
- Verificación de integridad

TCP proporciona, además de los servicios mínimos, los siguientes servicios:

- Entrega confiable
- Control de flujo
- Control de congestión

TCP es apropiado para casos donde no puede haber pérdida de paquetes (eg. mensajería) mientras que UDP es apropiado en casos donde dicha pérdida se puede tolerar (eg. streaming), ya que tiene menos overhead.

Dificultades encontradas

Concurrencia

Una de las dificultades que no podría faltar en el desarrollo con sockets es la concurrencia. El principal desafío era lograr que el servidor maneje varios clientes en paralelo (y cierre gracefully) y, a nivel capa de transporte, poder recibir desde el socket y enviar acknowledgments de manera asincrónica. Se pudo afrontar principalmente con el uso de Locks, Queues y Events.
El uso de concurrencia también dificulta mucho el debuggear, debido a posibles race conditions y situaciones que no se dan por igual en cada ejecución.

Dependencias

Irónicamente una de las dificultades que tuvimos al inicio era estructurar el proyecto para que funcionen correctamente las dependencias. Los módulos de python no se importan a partir del path relativo del archivo como otros lenguajes como C o JavaScript, lo cual nos atrasó un poco el comienzo.

Conclusión

Nuestra principal conclusión es que no somos lo suficientemente agradecidos con todo lo que hace TCP y las personas que lo diseñaron y programaron. Más allá de eso, podemos sacar algunas conclusiones de la tabla de comparación del socket Stop & Wait y Selective Repeat.

Para empezar, respecto al tamaño del archivo, es claro que independientemente del packet loss a más cantidad de bytes transferida más se nota la diferencia de velocidad entre Selective Repeat y Stop & Wait. Esto es así ya que para archivos pequeños Stop & Wait envía todo el contenido en unos pocos RTT, por lo que la diferencia no se nota tanto. En cambio, para archivos grandes, Stop & Wait ya se demora bastantes más RTT que Selective Repeat.

Respecto al packet loss, como es de esperar, cuantos más segmentos se pierdan más se demoran los protocolos. Stop & Wait se ve más afectado que Selective Repeat, ya que para cada segmento que se pierde se tiene que esperar un RTO entero. En cambio, Selective Repeat puede seguir enviando otros segmentos (si la ventana lo permite).