# Herramienta para el desarrollo y evaluación de métodos de control de congestión en TCP

**Guillermo Rigotti**
**UNICEN – Fac. de Ciencias Exactas-ISISTAN**
**Pje. Arroyo Seco, (7000) Tandil, Bs. As. Argentina**
**TE: +54-2293-439682 FAX: +54-2293-439681 Email: grigotti@exa.unicen.edu.ar**

**Abstract**

Since its definition, TCP has been adapted to significant changes that has took place in its environment. When TCP was first intoduced, the connected networks had few hosts and the communication links were homogenous. In few years, the Internet exponentially grew in number of hosts and networks. In addition, the communication technologies varied significantly. In spite of these changes, TCP has been adapted successful to fulfill their objectives: to obtain an efficient use of the network and to offer an suitable service to the users. This adaptation has been consequence of a considerable research activity, consistent mainly in the development of congestión control mechanisms This work, motivated by the constant necessity to adapt TCP to new conditions of operation, consists of the development of the basic functionality of TCP independently of any operating system, and using Java, a portable language. This development will be used as a tool for the development and evaluation of new heuristics related to congestion control in TCP. The new code could be including with little effort in the developed TCP, and the tests - exchange of information between two hosts through the Internet - will be facilitated due to the portability of the chosen platform.

**Keywords:** TCP, TCP congestion control, TCP performance.

**Resumen**

Desde que fue definido, el protocolo TCP se ha adaptado a un número considerable de cambios en su medioambiente de operación: definido en momentos en que la red subyacente estaba compuesta de vínculos homogéneos y pocos equipos conectados, ha visto crecer a la Internet de manera exponencial, y a las tecnologías de comunicación aumentar su capacidad y diversificar sus características de manera no menos significativa. Pese a estos cambios, TCP ha sido adaptado exitosamente para cumplir con sus objetivos de lograr un uso eficiente de la red y ofrecer un servicio adecuado a los usuarios. Esta adaptación ha sido consecuencia de una considerable actividad de investigación, consistente principalmente en el desarrollo de mecanismos de control de congestión. Este trabajo, motivado por el hecho de que TCP deberá aún seguir adaptándose a cambios considerables en su medioambiente, consiste en el desarrollo de la funcionalidad básica de TCP en un lenguaje portable, Java, y de manera independiente de cualquier sistema operativo. Este desarrollo se utilizará como herramienta para el desarrollo y evaluación de nuevas heurísticas de control de congestión en TCP. El nuevo código podrá ser incluido con poco esfuerzo en el TCP desarrollado, y las pruebas – intercambio de información entre dos equipos a través de la Internet - se verán facilitadas debido a la portabilidad de la plataforma elegida.

**Palabras claves:** TCP, control de congestión TCP, performance TCP.

# 1. Introducción

TCP (Transfer Control Protocol) [1] es un protocolo de nivel transporte que se originó junto con la arquitectura que lleva su nombre (TCP/IP), permaneciendo vigente en la actualidad. Provee un servicio de transferencia de datos orientado a la conexión y confiable, tratando a la información a enviar como un flujo continuo de bytes (modo stream). El servicio que provee se complementa con el provisto por UDP (User Datagram Protocol) [2], no confiable y no orientado a la conexión, cubriendo así un amplio espectro de los requerimientos de los usuarios.

Una caracteristica que debe destacarse de TCP es que ha sobrevivido hasta la actualidad sin modificaciones en su diseño, adaptándose a cambios significativos en su medioambiente de operación a través de sucesivas mejoras en sus mecanismos.
A continuación se mencionan brevemente algunos de dichos cambios y las versiones más relevantes del protocolo.
TCP fue originalmente definido en [1], alli se tratan sus aspectos básicos:
1-El formato de frame, que se ha mantenido hasta la actualidad sin modificaciones aunque se le han agregado nuevas opciones [3] ,2- La interfaz genérica con las aplicaciones, que define requerimientos básicos realizados por las aplicaciones (open, read, write, close, etc) que en la práctica se encuentra condicionada por el sistema operativo, y se implementan a través de la interfaz sockets, y 3- la operación del protocolo, definida como una maquina de 11 estados que se activa por eventos, los cuales pueden ser primitivas del nivel aplicación, segmentos recibidos del TCP remoto y vencimiento de timers. Los estados definidos contemplan la totalidad de la operación del protocolo, teniendo en cuenta en detalle los aspectos de establecimiento de conexión (three way handshaking) y terminacion de conexión. La transferencia de información, que se lleva a cabo fundamentalmente en el estado ESTABLISHED[1], no se trata dentro de la FSM, sino que por su simplicidad – en aquella primera especificación [2]– solo se definen ciertas variables propias de las partes emisora y receptora y algunas indicaciones sobre la operación.
El aspecto en que más ha variado TCP está relacionado con mejoras en el control de congestión, motivadas por el crecimiento de las redes; más tarde aparece el desafío de operar eficientemente sobre diversas tecnologías, tales como satélites y wíreless [4] [5].. Todo esto ha dado lugar a los cambios que brevemente se mencionarán a continuación.
En la versión original, no se realizan consideraciones acerca del control de congestión protocolo/red, debido a que la homogeneidad de las redes y la baja cantidad de equipos conectados no requería de este tipo de control (se especifica sólo un control de congestión emisor/receptor, basado en ventanas deslizantes de longitud variable). Estas caracteristicas las podemos encontrar en el TCP Berkeley (1983) que puede considerarse como la implementación del RFC 793.
A partir de 1988, en TCP Tahoe y posteriores, aparece un control de congestión orientado a no saturar la red, compuesto de dos mecanismos independientes pero que trabajan en conjunto en las implementaciones: Slow Start y Congestion Avoidance. Con estas modificaciones, el emisor, que antes podía inyectar paquetes hasta el límite de la ventana del receptor, pudiendo saturar la red, está ahora restringido por un limite adicional, que se obtiene de medir la respuesta de la red (considerando llegada de asentimientos) a la carga inducida. Ambos mecanismos se aplican al iniciar la transmisión, y luego cada vez que el emisor detecta síntomas de congestión. Mientras que Slow Start aumenta la tasa exponencialmente, Congestion Avoidance lo hace linealmente [6].

[1] Es posible transferir datos en los segmentos que se intercambian para el establecimiento de conexión, así como también, debido al esquema de desconexión asimétrica soportada, recibir datos luego de haber solicitado un CLOSE.
[2] En ese momento no se tuvo en cuenta el control de congestión.

Posteriormente se incorpora Fast retransmit, mejora que permite a un emisor retransmitir un paquete que detecta como perdido de manera inmediata, sin esperar el tiempo de vencimiento del timer de retransmisión, Está basado en la característica de TCP de emitir un ACK cada vez que llega un segmento. Por lo tanto, si un segmento se ha perdido, se comenzará a recibir ACKs duplicados. Se tiene en cuenta la posibilidad de recepción de ACKs duplicados como consecuencia de un reordenamiento temporal de paquetes en la red.
En 1990 aparece TCP Reno, que presenta las caracteristicas del Tahoe, más la incorporación del algoritmo Fast recovery, mejora que permite que el TCP emisor, ante la pérdida de un paquete reaccione con el mecanismo de Congestion Avoidance y no el de Slow Start, con lo cual no se baja drásticamente la tasa de transmisión. TCP asume que cuando la pérdida del paquete se detecta por la recepción de ACKs duplicados, no hay congestión en la red, ya que se reciben ACKs, y que la pérdida ha afectado sólo al paquete en cuestión [7].
Posteriormente aparecen modificaciones tendientes a tratar nuevos problemas, surgidos del aumento de la capacidad de la líneas de transmisión a las cuales se las denomina "long fat networks" (LFN) aludiendo a su gran capacidad de transmisión y a su considerable demora de propagación (en relación a su capacidad). Estas modificaciones, demasiado extensas para describir aquí, consisten entre otras cosas en la posibilidad de incrementar la longitud de los segmentos TCP, para explotar adecuadamente ese tipo de líneas, en el uso de ACKs selectivos (SACKs) para que el emisor retransmita sólo los paquetes no recibidos, en la medición más refinada de tiempos utilizada para estimar tiempos de retransmisión (timestamp option), etc. Para mayores detalles referirse a [8] [9] [10].
TCP Vegas [11] es una mejora que se caracteriza por incrementar el throughput y disminuir las pérdidas que sufre Reno a través de algoritmos mejorados. Sólo involucra cambios de implementación en el lado emisor, resultando compatible con cualquier otra versión válida de TCP.
Actualmente TCP continúa en evolución, no sólo para mejorar su performance, sino para enfrentar nuevos desafíos surgidos de su operación sobre nuevas tecnologías de transmisión.
En resumen, TCP se ha adaptado a importantes cambios en su medioambiente de operación: desde sus comienzos, operando sobre redes homogéneas con pocos equipos conectados, donde no era necesario el control de congestión entre el nivel de red y el protocolo, hasta la actualidad, operando en redes heterogéneas con gran cantidad de equipos en las cuales cobran importancia fundamental dichos mecanismos.
Como respuesta a esos cambios, TCP ha incorporando diferentes algoritmos de control de congestión, que han ido mejorando su performance y adaptándolo a los nuevos requerimientos. Además, la operación sobre tecnologías diversas, que implica la combinación de vínculos altamente confiables y de gran capacidad de transmisión (fibra óptica) con otros de menor capacidad y considerable tasa de errores (vínculos wireless), ha producido importante actividad de investigación tendiente a adaptar el protocolo a esas características.
Es precisamente esta continua actividad en torno a TCP, centrada principalmente en la mejora de su performance a través de nuevos algoritmos de control de congestión, lo que nos condujo a desarrollar el TCP descripto en este trabajo. Nuestra convicción es que esta implementación simplificada debido a que no debe ocuparse de problemas inherentes a su implantación en sistema operativo alguno y a que maneja solo una conexión, resultará de utilidad para la experimentación con el protocolo. Esta consistirá en la prueba de algoritmos ya implementados, intentando su modificación y mejora, con pruebas en escenarios reales[3]. Adicionalmente, esta implementación puede ser de utilidad en otros ámbitos relacionados con nuestros temas de investigación, como se describe en la sección destinada a las conclusiones.

[3] Pares de equipos conectados a través de Internet.

El resto del trabajo se organiza de la siguiente manera: en la sección 2 se enuncia el objetivo de este trabajo y se describen las características principales del mismo, las cuales permiten determinar los alcances y limitaciones del resultado obtenido. En la sección 3 se describe la estructura del TCP implementado, considerando los datos almacenados para una conexión y el proceso de los mismos; esta descripción se completa en la sección 4 con la descripción de la interfaz ofrecida a las aplicaciones. La seccion 5 contiene las conclusiones, trabajos en curso que continúan el descripto, y otros usos del prototipo desarrollado. Las referencias bibliográficas de mayor relevancia se citan en la sección 6.

## 2. Objetivo, alcances y limitaciones de la implementación

Resulta conveniente, antes de describir con mayor detalle el desarrollo realizado, especificar cuál fue el objetivo perseguido y aclarar cuáles son los alcances y limitaciones (comentados en la descripción de las características) fijados en función del primero.
El objetivo fundamental del presente trabajo es obtener una versión de TCP, tal como es definido en el RFC 793 [1], a la cual pueda agregársele de manera clara y modular, la funcionalidad de las diferentes versiones existentes de TCP (Tahoe, Reno, etc) , para evaluar su funcionamiento en condiciones reales, es decir, comunicando las dos partes del protocolo a través de la Internet. Posteriormente, se procederá a la eventual modificación de los mecanismos ya existentes y desarrollo de nuevas heurísticas, las cuales podrán ser codificadas y agregadas a este TCP sin mayor esfuerzo, para finalmente ser evaluadas en su ambiente real de operación.
A continuación se describen las características principales del desarrollo, remarcando los alcances y limitaciones del mismo.
1-Programación simple, modular y código fácil de modificar a quien desee incorporar nueva funcionalidad en el protocolo: es una característica que limita en cuanto a performance, ya que el uso de estructuras de datos y procesos que operen de manera eficiente, muchas veces disminuye la legibilidad del código y dificulta la incorporación de agregados y modificaciones. La decisión tomada, de acuerdo con el objetivo planteado, fue sacrificar la eficiencia en función de la claridad y modularidad.
2-Manejo de una única conexión por parte de TCP e imposibilidad de reutilizar la instancia TCP en caso de abortar o cerrar la conexión. La consecuencia de lo anterior es que una aplicación debe crear una instancia de TCP por cada conexión que desee realizar, en contraste con el TCP "real", que es capaz de manejar múltiples conexiones concurrentemente. Por otra parte, para simplificar el proceso, se resolvió no reinicializar la instancia TCP, lo que implica que una aplicación que cierre una conexión, si desea abrir otra, deberá crear una nueva instancia TCP. Se adoptó esta decisión debido a que para los objetivos planteados no es importante dicha característica, que contribuiría a complicar el código.
3-Portabilidad. Esta característica es importante en esta aplicación, ya que por su naturaleza, exige ser probada entre un par calquiera de equipos conectados a la Internet.
Dos aspectos fundamentales para lograr portabilidad son: la implementación del protocolo a nivel usuario, es decir, sin interferir con el sistema operativo, y la elección de un lenguaje de amplia difusión. Por este último motivo, el lenguaje elegido fue Java. Como se menciona en el siguiente párrafo, en un principio la elección de Java trae aparejado un problema relacionado con el acceso al nivel de red, que constituye el soporte de comunicación para TCP.
4-Uso de UDP como soporte de comunicación. Esta característica resulta contraria al uso de las facilidades de comunicación que debería hacer un protocolo de nivel transporte, es decir, uso de IP, que esta en el nivel de red. Se decide utilizar UDP, debido a que, al estar el TCP implementado en Java, no es posible disponer de librerías standard para el acceso al nivel de red. Estas no son provistas por Java debido a que para algunos, resultaría en un problema de seguridad, mientras que

otros argumentan que seria muy útil ya que pondría a Java a la altura de otros lenguajes como C++ en aspectos relativos a aplicaciones de monitoreo de red y similares. Si bien esta característica impide la comunicación de nuestro TCP con TCP "reales", ya que en nuestro caso el segmento TCP se encapsula en uno UDP y este en uno IP, desde el punto de vista de la evaluación, no representa mayores problemas ya que UDP es un protocolo que no interfiere en modo alguno con el envío de datos. La limitación que surge de esta característica es que se hace imposible comunicar nuestro TCP con versiones reales[4]. Sin embargo, previendo la necesidad de esta prueba, se utilizará algún soporte de amplia difusión para acceso al nivel de red, tal como libpcap [12] o Winpcap [13] desde ambientes Windows, con una librería de acceso desde Java (Jpcap [14]), a efectos de poder interactuar directamente con IP.
5-Preservación del formato de PDU (Protocol Data Unit) de TCP. El formato de segmento TCP se mantuvo inalterado. Se contempló también el proceso de opciones. Esta es una característica fundamental tanto para las pruebas realizadas como para cuando se realice comunicación con TCPs "reales".

# 3. Estructura del TCP desarrollado

Para permitir un fácil agregado de funcionalidad al TCP básico (RFC 793), se trató de preservar la simplicidad, lo cual condujo a definir los principales elementos componentes del protocolo de forma separada, en contraposición con una implementación que busque lograr eficiencia, en la cual se podrían fusionar algunos de ellos. Lo mismo puede decirse de los diferentes threads de ejecución, que se describen posteriormente. Una implementación que si bien no fue utilizada como modelo, pero fue de gran utilidad para entender los procesos descriptos en el RFC 793, puede encontrarse en [15].
En la figura 1 se muestran los componentes de la implementación que se describirán a continuación. Como puede apreciarse, los threads trabajan de manera independiente, comunicándose a través de eventos dirigidos a la FSM, mientras que ésta, cuando corresponde, utiliza señales de control (por ejemplo para indicar al thread de la aplicación que la información solicitada está disponible). Se muestra además el acceso a los datos almacenados en los diferentes objetos manejados por TCP.

## 3.1 Organización de los datos de una conexión

### *3.1.1 Ventana de emisión*

La ventana de emisión involucra dos elementos relacionados con el envío de datos: un buffer de un tamaño máximo prefijado, destinado a alojar los bytes que la aplicación solicita enviar, y la ventana de emisión propiamente dicha, que es el conjunto de bytes contiguos de dicho buffer que han sido enviados por TCP, pero de los cuales aún no se ha recibido confirmación de recepción. Por ejemplo, definir un buffer de 1000 bytes, con una ventana de emisión asociada de 100 bytes, significa que TCP puede aceptar hasta 1000 bytes entregados por la aplicación para ser enviados; éstos serán copiados en el buffer. Por otra parte, el tamaño máximo de la ventana de emisión es 100 bytes, lo que significa que en cualquier momento, no puede haber más de 100 bytes enviados de los cuales no se haya recibido confirmación. La ventana de emisión puede verse entonces como desplazándose sobre segmentos contiguos del buffer, de una longitud variable entre 0 y 100. Cuando la aplicación entrega datos a enviar, la cantidad de bytes en el buffer aumenta. Cada vez que se envía un grupo de bytes, el límite superior de la ventana de emisión se corre hacia números de secuencia más altos,

[4] Por "version real" entendemos un TCP implementado en un lenguaje tal como C++ y formando parte del kernel del sistema operativo.

mientras que cuando se recibe un ACK, los bytes confirmados son eliminados del buffer, permitiendo que la aplicación pueda entregar más, y a su vez son eliminados de la ventana de emisión, aumentando su límite inferior y permitiendo al TCP local enviar más al TCP remoto.
El tamaño del buffer de emisión está dado por la cantidad de memoria que TCP asigna a la conexión. En general, esto puede ser configurado por las aplicaciones, simplificándose en nuestro caso ya que TCP dispone de todos sus recursos para asignarlos a la única conexión que soporta. El tamaño de la ventana de emisión, estará dado, en cada momento, por el método de control de congestión (emisor/receptor y emisor/red) utilizado.
Mantiene variables relacionadas con el envío de datos, tal como se define en RFC 793 (ùltimo número de secuencia confirmado (SND_UNA), próximo número de secuencia a recibir (SND_NXT), número de secuencia inicial de la conexión (ISS), posición del puntero a datos urgentes (SND_UP), tamaño de la ventana de emisión (SND_WND) y las variables que almacenan los numeros de secuencia y confirmación de los segmentos que fueron utilizados para la última actualizacion de ventana (SND_WL1 y SND_WL2).
Se encarga también de memorizar la ubicación de la señal de FIN, ya que cuando la aplicación realiza un CLOSE, es posible que haya datos pendientes de envío, y por lo tanto se debe diferir el envío de FIN hasta que se envíe la totalidad de aquellos.
Las funciones básicas de esta ventana son
*int adddata(byte[] data)*, agrega datos a enviar al buffer de emisión y es invocada como consecuencia de una solicitud de envío de datos de la aplicación al TCP,
*byte[] getdata(int num)*, entrega la cantidad de datos solicitada para ser enviada, aumentando el tamaño de la ventana emisión. Es invocada por el thread de emisión, que entre sus tareas tiene la de armar segmentos con datos para enviar al TCP remoto.
*void ackreceived(long acknum)*, que actualiza la ventana de emisión y el buffer de emisión en función del ACK recibido.

### *3.1.2 Ventana de recepción*

De manera similar al caso anterior, existen aquí dos elementos relacionados: el buffer de recepción y la ventana de recepción. El primero es el espacio asignado a la aplicación por TCP, donde este almacena los datos recibidos hasta que la aplicación los lea, momento en el cual será liberado ese espacio del buffer. El tamaño de este buffer, que coincide con el tamaño máximo de la ventana de recepción, se configura como parámetro. La ventana de recepción puede considerarse como desplazándose sobre el buffer, sobre los lugares no ocupados, e indica los números de secuencia que pueden ser aceptados. Su tamaño varia, desde un máximo igual al del buffer, y disminuyendo en función de la ocupación del mismo. En el caso extremo en que la aplicación no lea al ritmo de llenado del buffer, su tamaño puede disminuir a cero.
Se mantienen aquí las variables definidas en RFC 793 relacionadas con la recepción:
número de secuencia del próximo carácter a recibir (RCV_NXT), número de secuencia del puntero a datos urgentes (RCV_UP), número de secuencia inicial de la conexión en recepción (IRS) y tamaño de la ventana (RCV_WND)
Además, se almacena aquí un eventual FIN recibido, ya que no siempre es posible procesarlo junto con el segmento que lo contiene, debido a que puede haber segmentos previos perdidos que impiden dicho proceso
Las funciones básicas de este elemento son
*void addsegment(long seqini, byte[] data)*, recibe como parámetro un conjunto de bytes provenientes de un segmento recibido y el número de secuencia del primer byte y agrega aquellos a la ventana de recepción, teniendo en cuenta sus limites. Como consecuencia del agregado, modifica las variables correspondientes. Un chequeo adicional que se realiza aquí es aquel que permite determinar si como consecuencia de la incorporación de los datos nuevos, la eventual marca de FIN

(recibida en este segmento o en otro, si este sufrió reordenamieto o pérdida y se trata de una retransmisión) e informar a la FSM para que cambie al estado correspondiente y envíe el ACK a

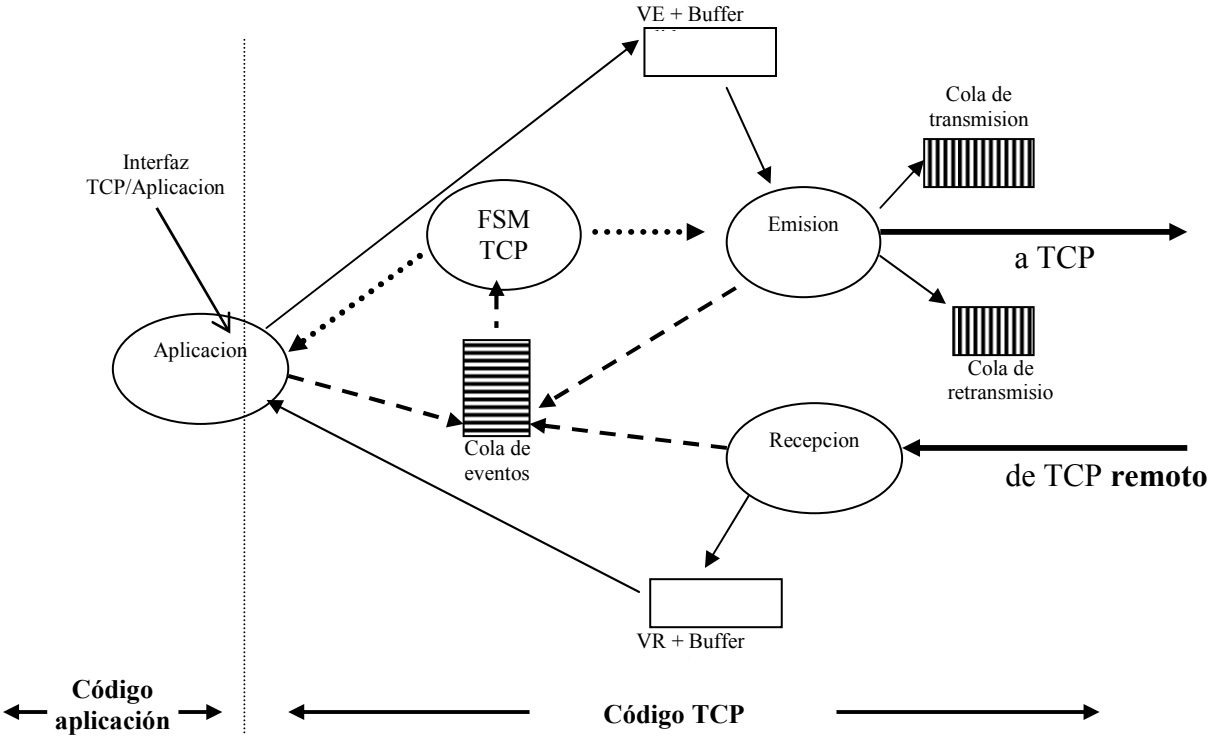


*Fig 1. Estructura del TCP implementado. Se muestran los threads y los datos comunes utilizados. Los threads se representan mediante elipses; el intercambio de datos con íinea llena, los eventos con íinea cortada, y la información de control con*

dicho FIN. Esta parte del código es la que se encarga de programar el envío del ACK correspondiente si el límite de la ventana ha cambiado.
*byte[] getchars(int num)*, saca de la ventana (si es posible) la cantidad de caracteres contiguos solicitada, y actualiza el límite inferior.
*boolean is_acceptable(TCPSegment seg)*, determina, dado un segmento TCP, en función de su número de secuencia y longitud, y de acuerdo con lo especificado por el RFC 793, si el segmento es válido o se trata de un duplicado atrasado no válido correspondiente a una instancia anterior de la conexión.

### *3.1.3 Colas de transmisión y retransmisión*

En ellas son colocados los segmentos que deben ser enviados al TCP remoto por primera vez, o como consecuencia de una retransmisión. La cola de transmisión es alimentada por la FSM en caso en que deba generar un segmento de control (por ejemplo un RST) y por el thread de envío, cuando extrae bytes de la ventana de emisión para generar los segmentos de datos correspondientes. Es a su vez este thread el que se encarga de transmitir los segmentos de la cola, en orden. La cola de retransmisión recibe segmentos por parte del thread de envío, quien los coloca en ella cuando son transmitidos por primera vez, siempre que dicha transmisión este sujeta a reintentos. Una vez en la cola de retransmisión, las retransmisiones son disparadas por timers asociados a los segmentos. Esta

cola es modificada cuando se recibe un ACK, eliminando los segmentos confirmados. Los segmentos también se eliminan cuando se cumple el número máximo previsto de retransmisiones, provocando que TCP produzca la excepción correspondiente..
Los elementos almacenados en estas colas contienen además del segmento a enviar, información adicional entre la cual se incluye referencia a los timers asociados y a la cantidad de reintentos restante.

### *3.1.4 Cola de eventos*

Esta cola contiene objetos que representan los eventos que debe procesar la FSM. Cada evento está compuesto por un código que identifica su tipo, y un objeto que contiene la información correspondiente. Por ejemplo, un evento de tipo ARRIVE (llegada de un segmento) contiene un objeto TCPSegment (un segmento TCP).
Los eventos son colocados en la cola por los diferentes threads: el thread de recepción coloca los arribos de segmentos, los threads de la aplicación, ejecutando código TCP, los eventos que reflejan los requerimientos de aquélla, por ejemplo read, write, etc. El propio thread de la FSM y los timers también pueden generar eventos. El orden de atención de los mismos por parte de la FSM es FIFO..

## 3.2 Proceso

El proceso realizado se lleva a cabo a través de varios threads que operan asincrónicamente, comunicándose a través de objetos compartidos: las ventanas de emisión y recepción, las colas de transmisión y retransmisión y la cola de eventos de la FSM. Los threads involucrados se describen a continuación.

### *3.2.1 Thread de la FSM TCP*

Este thread representa la FSM TCP, tal como se define en el RFC 703. Está implementado, por razones de claridad y eficiencia, como un doble switch, a través del cual se procesan los eventos en función del estado corriente de la FSM. Los estados definidos son los 11 especificados en el RFC 793, al igual que los eventos. Los primeros abarcan la fase de establecimiento de conexión, intercambio de información y finalización de la conexión, mientras que los eventos pueden provenir de acciones solicitadas por la aplicación, de vencimiento de timers o de segmentos provenientes del TCP remoto.
El código se trató de mantener simple, a través del uso de procedimientos comunes a diferentes combinaciones estado/evento. Debe destacarse que se sigue estrictamente la especificación del RFC 793, ya que esto facilita la comprensión del código y la facilidad para el agregado de módulos.
Se definió un evento adicional, propio de la implementación, que permite realizar el reset de la FSM, informando adecuadamente a la aplicación.
El proceso realizado consiste en un ciclo en el cual se toma el próximo evento en la cola de eventos, y en base al estado corriente se lo procesa, generando solicitudes de envío de segmentos, nuevos eventos y cambios de estado. El proceso de un evento se completa antes de procesar el siguiente.

### *3.2.2 Thread de recepción*

Por el momento, este thread es el mas simple[5], se encarga de inicializar el socket UDP, que constituye el soporte de comunicación para nuestro TCP, y posteriormente, durante el resto de la vida del proceso, de leer los bloques UDP que arriban provenientes del lado remoto, entregando a la FSM TCP, en forma de eventos de tipo ARRIVE, los segmentos TCP que aquellos contienen.

[5] En el futuro, podrá incluir chequeos adicionales no implementados actualmente, tales como verificación de checksum.

### *3.2.3 Thread de emisión*

Se ocupa de diversas tareas relacionadas con el envío de segmentos TCP. Se encarga de la inicialización y posterior mantenimiento de las colas de transmisión y retransmisión. Para eso se definen dos métodos, uno que permite agregar segmentos a la cola de transmisión, y otro que, al recibirse un ACK, permite eliminar de la cola de retransmisión los segmentos totalmente confirmados.
En su ciclo, cumple dos tareas diferentes:
Por un lado, chequea la ventana de emisión y en caso de ser posible (si existen datos a enviar y si la ventana de recepción del otro TCP y la de congestión no son cero), construye los segmentos a enviar y los coloca en la cola de transmisión. En esta etapa es responsable de generar la indicación de FIN (en un segmento con datos o en uno destinado sólo a ese efecto) en caso de que la aplicación haya realizado un CLOSE.
Por otra parte, chequea el estado de la cola de transmisión, y en caso de encontrar segmentos preparados para ser transmitidos, los envía, colocándolos también en la cola de retransmisión, en caso de que se especifiquen reintentos. En este caso también se encarga de programar el primer vencimiento de timer para cada segmento.
Debe tenerse en cuenta que en la cola de transmisión puede haber segmentos a enviar que no han sido generados por este thread, sino que provienen de otros, por ejemplo de la FSM TCP, en caso de que tenga que generar una confirmación (ACK) sin usar piggybacking..

### *3.2.4 Threads de la aplicación*

Cuando un thread definido por la aplicación invoca a una función TCP, definida en la interfaz TCP-aplicación, ejecuta código TCP que se encarga de comunicar este thread con los que componen TCP, a través de colas. Por ejemplo, si un thread de la aplicación realiza una invocación a la función OPEN, esto provocará que el código TCP produzca el evento correspondiente para que sea procesado por la FSM, y mientras se espera una respuesta – las funciones son bloqueantes para la aplicación – que dependerá de la reacción del TCP remoto, el thread del usuario permanece bloqueado en el código de TCP, a la espera de la misma.
Normalmente las aplicaciones tendrán – entre otros posibles - un thread de recepción, que continuamente está invocando a la funcion read – bloqueante – de TCP, y otro de recepción, que invoca a write – tambien bloqueante -.

## 3.3 Timers

El mecanismo de manejo de timers se realizó de manera simple, utilizando las facilidades provistas por Java. Por el momento, es creado un objeto TimerTask por cada nuevo timer que se arranca. Si bien esto puede ser mejorado, utilizando un único timer general, no se lo hizo considerando el objetivo de mantener una mplementación simple y modular.
Se implementan los siete timers definidos por TCP: para controlar los tiempos máximos de espera de una reacción del otro TCP, para el mecanismo de envío de datos (piggybacking y retransmisión), para determinar si el TCP remoto aun está funcionando, y para la finalización de la conexión. Adicionalmente, la implementación define otro para que la aplicación pueda especificar en ciertos casos el tiempo máximo a esperar por una respuesta del TCP local. Para una explicación detallada de las funciones de los timers, referirse a [16].

# 4. Interfaz TCP/Aplicación

*4.1 Características de las aplicaciones*

Una decisión tomada inicialmente, junto con la elección de Java como lenguaje, fué la de hacer compatible el TCP a desarrollar con aplicaciones escritas en Java. Esto por supuesto no debe entenderse como que una aplicación Java escrita para utilizar TCP pueda utilizar, sin cambios, nuestro TCP, pero sí con mínimos cambios y manteniendo la estructura de dicha aplicación.
El objetivo buscado en este aspecto, es implementar una interfaz con las mismas funciones que provee Java 2 v 1.2.2 [17] para TCP. Este aspecto aún no ha sido completado, implementándose sólo la funcionalidad básica que permite establecer y terminar una conexión, e intercambiar datos. El resto de la funcionalidad se piensa implementar en breve plazo.
Debe destacarse sin embargo, que la mecánica de comunicación entre la aplicación y TCP, se encuentra totalmente definida y es utilizada por el subconjunto de las funciones que ya se encuentran implementadas.
Las invocaciones a TCP son bloqueantes, produciendo un retorno exitoso con la entrega de la eventual información solicitada. Los posibles errores que puedan producirse (dirección errónea al establecer conexión, datos nulos, tiempos vencidos, etc) son comunicados a la aplicación a través del mecanismo de excepciones provisto por Java. En este caso en particular, se define una nueva excepción, TCPException, que contempla las situaciones anormales especificadas en el RFC 793 y otras propias de la implementación.

### 4.2 Funciones implementadas

Las invocaciones implementadas hasta el momento son las siguientes
*byte[] read(int cantidad) throws TCPException;* Devuelve el número de bytes especificado. Si se especifica 0 devuelve todos los que hay.
*void Aopen() throws TCPException;* Abre la instancia TCP en modo activo (open activo)
*void Popen() throws TCPException*; Abre la instancia TCP en modo pasivo (open pasivo)
*void write(byte[] data) throws TCPException*; Envía un conjunto de bytes por la conexión
*void close() throws TCPException*; Produce el cierre ordenado de la conexión.

Por el momento, antes de realizar cualquier operación sobre TCP, debe ser creada una instancia del mismo, invocando a su constructor. Esta es una diferencia con la interfaz provista por Java, que será resuelta a breve plazo.

### 4.3 Mecanismo de invocación

Como se mencionó, la invocación de la aplicación produce que el código TCP genere un evento a la FSM , y quede esperando una respuesta de la misma, para retornar este resultado a la aplicación. Se debe hacer una distinción entre los diferentes tipos de llamada, de acuerdo a si el resultado solicitado puede ser devuelto a través de la ejecución del evento correspondiente por parte de la FSM, o esta respuesta depende de factores externos, por ejemplo, respuesta del TCP remoto. En cualquier caso, no es posible demorar a la FSM en un evento, esperando por algún resultado. El código TCP invocado por la aplicación, ejecutado por el thread de invocación, es el que se encarga de implementar el mecanismo de comunicación entre la FSM y la aplicación, manteniendo la independencia de la FSM respecto de la aplicación, y posibilitando que las llamadas de la aplicación sean bloqueantes, al estilo de Java.
Debemos distinguir los siguientes casos

En el más simple, la llamada se resuelve completamente durante el proceso del evento correspondiente por parte de la FSM. Por ejemplo, en el caso de Status – no implementado aún – que consiste en copiar las variables relevantes de TCP para entregarlas a la aplicación-.
En otros casos, el evento producido puede o no dar una respuesta a la aplicación. Por ejemplo, en el caso de un read, es posible que cuando se genera el evento correspondiente, la FSM tenga la cantidad de caracteres requerida, que se devuelve a la aplicación, con lo cual finaliza la invocación con el fin del proceso del evento. Sin embargo, podría ocurrir que la cantidad de bytes solicitada no estuviera disponible, y como el llamado es bloqueante, se debería esperar que se tenga dicha cantidad de datos en el buffer o que venza el timer de tiempo máximo de espera de la aplicación por el TCP. En ninguno de estos casos es posible demorar el evento de la FSM hasta que se produzca alguna de las alternativas.. La solución que se adopta consiste en que el código invocado por la aplicación quede en un ciclo, en el cual realiza repetidas invocaciones a la FSM, a intervalos regulares, hasta que se tiene éxito o se produce una condición de error. De esta manera, los eventos se resuelven a ritmo de la FSM, sin interferir con otros que deba procesar. Los posibles casos aquí serian la recepción de los datos, retorno normal, o un error de conexión o vencimiento de timer, lo que produce que el código TCP invocado por el thread de la aplicación genere una TCPException.
Otro caso lo constituye el de un Open, ya que aquí hay un único evento producido como consecuencia de la aplicación, pero debe esperarse por una reacción del otro TCP. No es posible ni tiene sentido generar repetidos eventos open, sino que se debe memorizar el evento original, para luego poder entregar el resultado. Para lograr esto, se introduce una variable en el código de la FSM, que recuerda el evento open; luego, cuando se recibe el evento correspondiente, que determina si el open fue o no aceptado, la misma FSM lo anunciara al código de invocación de TCP, quien procederá de la misma manera que fue descripto en el caso anterior. Debe notarse que es posible tener múltiples invocaciones concurrentes en el caso de un read o write, pero se permite una única invocación a open[6].

## 5. Conclusiones y continuación del trabajo

Lo presentado en este paper constituye la primera etapa de un proyecto cuyo objetivo final es la evaluación de los mecanismos de control de congestión existentes en TCP, y la implementación de nuevas heurísticas que permitan mejorar y adaptar TCP a nuevos medios de comunicación y/o nuevos requerimientos de las aplicaciones. La meta fijada en esta primera etapa, el desarrollo de un TCP portable con un diseño que permite introducir fácilmente nuevos mecanismos, ha sido alcanzada satisfactoriamente.
Es de destacar que debido a la información de carácter general contenida en el RFC 793, fue necesario recurrir a implementaciones “reales” en ciertas ocasiones. Sin embargo, estas ultimas sólo fueron de utilidad para comprender los mecanismos de TCP, ya que el código, contrariamente al de esta implementación, presenta una complejidad considerable debido a la necesidad de lograr un proceso eficiente y a su interacción con el sistema operativo.
La implementación demandó un volumen considerable de trabajo, pese a haber sido simplificada de la manera expuesta. Aunque resta realizar ajustes en el código, la performance es aceptable, obteniéndose tiempos de transferencia del orden de los obtenidos utilizando el soporte TCP provisto por Java[7]. De acuerdo con lo previsto, las aplicaciones pueden adaptarse al nuevo soporte con mínimas modificaciones.
En lo inmediato, el trabajo será continuado con la implementación de una interfaz completa, que provea toda la funcionalidad TCP ofrecida por Java. En paralelo, se probará exhaustivamente el

[6] Esto es controlado por la FSM, al analizar el par estado/evento.
[7] Las pruebas se realizaron en condiciones de transmisión ideales, sin congestión ni errores, debido a que el prototipo implementado aun no incluye los mecanismos de control de congestión

funcionamiento del protocolo confrontando su performance con la del TCP ofrecido por Java. La conclusión de estas tareas permitirá continuar el trabajo incluyendo los diferentes métodos de control de congestión. Adicionalmente, mediante el uso de facilidades de acceso al nivel de red, se estará en condiciones de probar la comunicación de nuestro TCP con cualquier otra versión real.
En una última etapa, se intentará el desarrollo de nuevas heurísticas que mejoren la performance del protocolo.
Adicionalmente, este protocolo podrá servir de punto de partida para el desarrollo de protocolos portables operando sobre UDP, con características orientadas a aplicaciones particulares, por ejemplo con un control de envío de los datos por parte de la aplicación, con diferentes métodos de chequeo de errores, etc.

## 6. Bibliografía


[1] Postel, J. "Transmission Control Protocol, DARPA Internet Program Protocol Specification", RFC 783, 1981
[2] Postel, J. "User Datagram Protocol", RFC 768, 1980
[3] Pentikousis, K, Hadr, H. "Quantifying the Deployment of TCP Options", IEEE Communication Letters, April 2004.
[4] Balakrishnan, N. Padmanabhan, S Seshan and. Katz, "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links", Computer Science Division, Department of EECS, University of California at Berkeley
[5] M. Allman, Ed., S. Dawkins, D. Glover, J. Griner, D. Tran, T. Henderson, J. Heidermann, J,, Touch, H, Kruse, S Ostermann, K. Scout, J. Semke"Ongoing TCP Research Related to Satellites", RFC 2760, February 2000
[6] Stevens, W. "TCP Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery Algorithms", RFC 2001, 1988
[7] Floyd S. Henderson T., "The NewReno modifications to TCP`s Fast Recovery Algorithm", RFC 2582, April 1999.
[8] V. Jacobson, R. Braden, "TCP Extensions for Long-Delay Paths", RFC 1072, October 1988
[9] V. Jacobson, R. Braden, L. Zhang, "TCP Extension for High-Speed Paths", RFC 1185, October 1990
[10] V. Jacobson, R. Braden, D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.
[11] Brakmo, L, O`Malley, S. and Peterson, L., "TCP Vegas: New Techniques for Congestion Detection and Avoidance", .Proc. of ACM SIGCOMM, October, 1994.
[12] libpcap Official Site, http://www.tcpdump.org/
[13] Winpcap Official Site, http://www.winpcap.org/
[14] Jpcap Official Site, http://sourceforge.net/projects/jpcap/
[15] Stevens, W, " TCP/IP Illustrated. Volume II: the Implementation", Addison Wesley, 1994.
[16] Stevens, W, " TCP/IP Illustrated. Volume I: the Protocols", Addison Wesley, 1994.
[17] JavaTM 2 Platform, Standard Edition, v 1.4.2, http://java.sun.com/j2se/1.4.2