UNIVERSIDAD DE EL SALVADOR
FACULTAD DE INGENIERIA Y ARQUITECTURA
ESCUELA DE INGENIERIA ELECTRICA

"Simulación Completa del Protocolo TCP Bajo Ambiente Ptolemy"

TRABAJO DE GRADUACIÓN PRESENTADO POR:

EDUARDO ENRIQUE CRUZ CARRILLO

PARA OPTAR AL TITULO DE:

INGENIERO ELECTRICISTA

CIUDAD UNIVERSITARIA, JULIO DEL 2001.

UNIVERSIDAD DE EL SALVADOR

AUTORIDADES UNIVERSITARIAS

RECTORA:

DRA. MARIA ISABEL RODRÍGUEZ

SECRETARIA GENERAL:

LICDA. LIDIA MARGARITA MUÑOZ VELA

FACULTAD DE INGENIERIA Y ARQUITECTURA

DECANO:

ING. ALVARO ANTONIO AGUILAR ORANTES

SECRETARIO:

ING. SAUL ALFONSO GRANADOS

ESCUELA DE INGENIERIA ELECTRICA

SECRETARIO:

ING. LUIS ROBERTO CHEVEZ PAZ

UNIVERSIDAD DE EL SALVADOR
FACULTAD DE INGENIERIA Y ARQUITECTURA
ESCUELA DE INGENIERIA ELECTRICA

TRABAJO DE GRADUACIÓN PREVIO A LA OPCION DE:

INGENIERO ELECTRICISTA

"Simulación Completa del Protocolo TCP Bajo Ambiente Ptolemy"

PRESENTADO POR:

BR. EDUARDO ENRIQUE CRUZ CARRILLO

COORDINADOR:

ING. CARLOS EUGENIO MARTINEZ CRUZ

ASESOR:

ING. CARLOS EUGENIO MARTINEZ CRUZ

CIUDAD UNIVERSITARIA, JULIO DEL 2001.

TRABAJO DE GRADUACIÓN APROBADO POR:

COORDINADOR:

ING. CARLOS EUGENIO MARTINEZ CRUZ

ASESOR:

ING. CARLOS EUGENIO MARTINEZ CRUZ

ACTA DE CONSTANCIA DE NOTA Y DEFENSA FINAL

En esta fecha Sábado 14 de julio de 2001 en la Sala de Lectura de la Escuela de Ingeniería Eléctrica, a las diez horas, en presencia de las siguientes autoridades de la Escuela de Ingeniería Eléctrica de la Universidad de El Salvador:

Firma:

1- Ing. Luis Roberto Chévez Paz
Secretario

ESCUELA DE INGENIERIA ELECTRICA
FACULTAD DE INGENIERIA
Y ARQUITECTURA
Universidad de El Salvador

Y, con el Honorable Jurado de Evaluación integrado por las siguientes personas:

Firma:

1- Ing. Carlos René Pérez Ramos

2- Ing. Werner David Meléndez

Se efectuó la Defensa Final reglamentaria del Trabajo de Graduación:

"Simulación completa del protocolo TCP bajo ambiente Ptolemy"

A cargo del Bachiller:

CRUZ CARRILLO, EDUARDO ENRIQUE

Habiendo obtenido el presente Trabajo una nota final global de: 8.1

(OCHO PUNTO UNO)

DEDICATORIA

El presente trabajo de graduación lo dedico principalmente a DIOS TODOPODEROSO, quien ha estado siempre a mi lado y gracias a su presencia y ayuda, que siempre se manifestó también a través de todos mis seres queridos y amigos, nunca me sentí solo a lo largo de mi esfuerzo por culminar mi carrera.

A MIS PADRES: Miguel Angel Cruz Cruz y Marta Carrillo de Cruz. Quienes me apoyaron incondicionalmente en todo y gracias a su esfuerzo, sacrificio y comprensión ahora puedo disponer de una profesión. Ellos formaron una de las partes mas importantes y fundamentales de mi triunfo y deseo compartir con ellos mi alegría como algo muy especial que me brindaron.

A MIS HERMANAS: Irma Violeta y Doris Guadalupe, porque nunca me desanimaron, sino por el contrario, siempre me brindaron su total ayuda en todos los sentidos, tesoro que siempre llevaré en mi corazón.

A MIS AMIGOS: Quienes fueran también mis compañeros, y especialmente los que estuvieron conmigo hasta el final, porque su apoyo y consejos me sirvieron de vitamina para poder continuar en los momentos mas difíciles de mi carrera.

Eduardo Enrique Cruz Carrillo.

PREFACIO

Dentro de los sistemas de comunicaciones actuales, existe como una gran revolución el uso de las computadoras conectadas a sistemas en redes de área amplia para lograr establecer conveniente y oportunamente el enlace de comunicación a través del globo terrestre. Esto demanda que tales sistemas, sean confiables y garanticen la calidad de la información que se transporta a través de las redes.

El desarrollo del TCP (Protocolo de Control de Transporte), y sus posteriores estudios han logrado encaminar esa difícil tarea de manera tal que su función juega uno de los papeles mas importantes dentro de un modelo mas amplio de capas de protocolos, los cuales al interactuar conjuntamente consiguen llevar a cabo el gran objetivo de comunicar a la humanidad.

Por ese y otos motivos, el presente trabajo trata de brindar una visión enfocada al mecanismo de operación del Protocolo de Control de Transporte (TCP) mediante un estudio teórico, el cual se complementa posteriormente a través del desarrollo de un simulador TCP por computadora, auxiliándose para ello de una de las herramientas mas versátiles que se encuentran disponibles dentro del campo de investigaciones en Ingeniería eléctrica e informática de la Universidad de California como lo es el software denominado PTOLEMY. El objetivo es poner a disposición el uso del simulador para que pueda servir como una herramienta mas de laboratorio dentro de un ambiente de software muy interactivo. El proyecto Ptolemy es dirigido por el profesor Edward Lee, y se ha nombrado así en honor a Claudius Ptolemaeus, astrónomo y matemático griego del segundo siglo.

INDICE

CAPITULO 1

"INTRODUCCION AL SIMULADOR TCP"

1.1 Introducción 1

1.2 Simplificaciones dentro del Simulador..... 2

1.3 Servicios Provistos por el TCP..... 3

1.3.1 Ventana Deslizante..... 3

1.3.2 Manejo de la Ventana Tonta.....4

1.3.3 Arranque Lento..... 4

1.3.4 Evasión de Tráfico..... 6

1.3.5 Retorno Exponencial y Estimación de Viaje Redondo.....8

1.3.6 Retransmisión..... 10

1.3.7 Retransmisión Rápida..... 11

1.3.8 Recuperación Inmediata.....11

1.3.9 Reconocimientos Retardados.....12

1.4 Conclusión..... 12

CAPITULO 2

"EL PROTOCOLO DE CONTROL DE TRANSPORTE TCP"

2.1 Introducción 14

2.2 Características del servicio de Entrega Confiable..... 14

2.3 Proporcionando Confiabilidad 17

2.4 Protocolo de Control de Transporte 22

2.5 Segmentos, Flujos y Números de Secuencia 22

2.6 Tamaño Variable de Ventana y Control de Flujo 24

2.7 Formato del Segmento TCP 25

2.8 Opción de Tamaño Máximo de Segmento 26

2.9 Computo de la Suma de Verificación 27

2.10 Acuses de Recibo y Retransmisión 28
2.11 Tiempo Límite y Retransmisión 28
2.12 Conclusión 31

CAPITULO 3
"EL DOMINIO DE EVENTOS DISCRETOS (DE) DE PTOLEMY"
3.1 Introducción 33
3.2 Un Vistazo a las Estrellas del Dominio "DE" Utilizadas en Simulación..... 33
3.3 Estrellas Fuente.. 35
3.4 Estrellas de Sumidero (sink) 38
3.5 Estrellas de Control 39
3.6 Colas, Servidores y Retardos..... 41
3.7 Conclusión..... 43

CAPITULO 4
"EL SIMULADOR TCP"
4.1 Introducción 45
4.2 El Transmisor TCP..... 45
4.2.1 MakeTCPPacket..... 46
4.2.2 SlidingWindow..... 48
4.2.3 TCPSender..... 50
4.3 El Receptor TCP..... 53
4.3.1 TCPRec..... 54
4.3.2 ReceiverWin..... 57
4.3.3 TCP_Dump..... 59
4.5 Modelo de Red..... 60
4.6 Conclusión..... 61

CAPITULO 5

"RESULTADOS DE SIMULACIÓN"

5.1 Introducción 62

5.2 Resultados Bajo Condiciones Normales..... 62

Para el Transmisor..... 62

Para el Receptor..... 63

Para el Canal de Transmisión (red) 63

5.3 Resultados con Retransmisión..... 73

Para el Transmisor..... 73

Para el Receptor..... 74

Para el Canal de Transmisión (red) 74

5.4 Conclusión..... 84

BIBLIOGRAFÍA..... 85

REFERENCIAS DE INTERNET..... 86

ANEXO A: PROGRAMACIÓN DE LAS ESTRELLAS

ANEXO B: EL ALGORITMO DE KARN

CAPITULO 1

"Introducción Al Simulador TCP"

1.1 INTRODUCCIÓN.

A Pesar de existir muchos simuladores para el protocolo TCP dentro del área de la comunicación de datos, se ha decidido implementar uno mas. El hecho de simular el protocolo TCP con Ptolemy puede no ser fácil de integrarlo como una herramienta que se pueda manejar como una fuente de tráfico general, pero puede formar parte de alguna galaxia muy utilizada dentro de Ptolemy.

Esta versión básica proporciona al usuario el mecanismo de control basado en la ventana TCP usual, el arranque lento, los algoritmos de evasión de tráfico y de estimación de viaje redondo.

Para verificar el simulador y poder observar su desempeño, se ha elegido un modelo topológico de red, el cual ya ha sido empleado por otros simuladores en el pasado y que ha demostrado el comportamiento fiel de una red con tráfico de datos. Con esto se puede comprobar la eficiencia del circuito y el desempeño del protocolo TCP bajo plena operación.

El simulador TCP presentado aquí fue concebido principalmente para mejorar el desempeño de los servidores y se logró sin alterar el protocolo en si, ni añadir ningún otro algoritmo ajeno al TCP. Desde el momento en que consideramos a los servidores como perfectos, es decir que no toman tiempo para el procesamiento de los paquetes o ejecución de los algoritmos de control, podemos idealizar algunos aspectos de operación. Los tamaños de ventana y números de secuencia están representados como valores enteros dentro del simulador, estas opciones están ya contenidas implícitamente dentro de la

implementación. Como el propósito principal es implementar un modelo básico TCP que incluya las características principales del algoritmo de control de trafico sin desperdiciar tiempo en mejoras no estandarizadas u otras extensiones innecesarias, se eligió simular esta versión que incluye explícitamente los servicios principales del algoritmo de control de trafico que usa el TCP.

1.2 SIMPLIFICAIONES DENTRO DEL SIMULADOR.

Las principales simplificaciones que se han hecho al sistema son las siguientes:

- En lugar de utilizar el reloj usual del TCP con una determinada resolución, el tiempo de información fue tomado de la marca de tiempo que va en los paquetes. Esta marca de tiempo es asignada a cada uno de los paquetes por Ptolemy, ya que en el dominio de los eventos discretos del software, utiliza estas marcas de tiempo para determinar la secuencia correcta dentro del listado de dichos eventos.

- Todos los paquetes poseen la misma longitud, la cual es la máxima longitud de segmento MSS[1].

- Un paquete está constituido únicamente por un encabezado el cual contiene campos describiendo las direcciones tanto del emisor y el receptor, el tamaño del paquete, el número de secuencia del paquete, el número de secuencia de byte y un campo para los acuses de recibo.

- Cuando se recibe un segmento fuera de orden, es decir fuera de secuencia, el TCP genera un acuse de recibo del último octeto (byte) recibido correctamente y el segmento es capturado en el buffer receptor local del TCP. Como también pueden generarse acuses de recibo duplicados debido al re ordenamiento de los paquetes, el transmisor espera que lleguen tres acuses de recibo antes de retransmitir el paquete

[1] MSS que en ingles quiere decir Maximun Segment Size

faltante. Al recibir el paquete que se había perdido, el receptor no reconoce al instante dicho paquete, pero si lo hace con el resto de segmentos que se han guardado en el buffer local, y esos segmentos almacenados pueden ser utilizados por el usuario. Si se pierde mas de un paquete solamente se reconocerá al paquete retransmitido así como todos aquellos paquetes recibidos correctamente. A parte de todo eso, el emisor tendrá que retransmitir el resto de paquetes. Como todos los segmentos que fueron enviados después del paquete perdido se reconocieron, el transmisor puede fácilmente enviar los datos empezando en el número de octeto al cual había llegado antes de la pérdida del paquete.

- Siempre existe una conexión, es decir que no hay necesidad de establecer una fase de conexión y terminación tal y como lo realiza el TCP en la práctica.

1.3 SERVICIOS PROVISTOS POR EL TCP.

A continuación se introduce una breve descripción de los servicios provistos por el simulador que aquí se ha implementado:

1.3.1 Ventana Deslizante:

A la llegada de los paquetes de datos en el receptor, se genera un acuse de recibo para el último byte recibido correctamente. En este reconocimiento va incluida información sobre el monto de datos que el emisor pudo seguir transmitiendo sin saturar al receptor, la llamada ventana de advertencia (*advertised window)*. Ahora el emisor puede mantenerse enviando mas datos hasta que le número de secuencia del último byte enviado sea igual a la suma del último byte reconocido mas el tamaño de la ventana de advertencia.

1.3.2 Manejo de la Ventana Tonta:

El mecanismo de manejo de la ventana tonta se usa para evitar el intercambio de los segmentos de datos pequeños. Esto puede darse por advertencia de ventanas pequeñas en vez de esperar hasta que pueda advertirse una ventana mas grande. En el lado del transmisor esto puede ocurrir debido al envío de pequeños segmentos en lugar de esperar por acuses de recibo adicionales. Cuando el tamaño de la ventana de advertencia cae debajo de MSS entonces se advierte una ventana de tamaño cero. Luego se envía un paquete de actualización cuando el tamaño de la ventana de advertencia llega a la mitad del tamaño de ventana máximo.

1.3.3 Arranque Lento:

El algoritmo de arranque lento añade otra ventana local al emisor TCP, una ventana de congestión (cwnd) que se mide en segmentos. Cuando se pierde un paquete o cuando se establece una nueva conexión, ésta ventana es puesta al tamaño de un segmento. A cada momento que se recibe un acuse de recibo o reconocimiento se incrementa la cwnd con el tamaño del segmento reconocido. El emisor puede ahora transmitir hasta un mínimo de cwnd mas la ventana de advertencia.

Todo TCP iniciaría una conexión con el emisor inyectando segmentos múltiples en la red, hasta el tamaño de la ventana de advertencia emitida por el receptor. Mientras todo esto progrese bien, ambos servidores se encuentran dentro de la misma LAN, pero si hay enrutadores y enlaces lentos entre emisor y receptor, los problemas pueden aumentar. Algún enrutador intermedio debe enfilar en cola los paquetes, y hasta es posible para ese enrutador contar sin espacio.

El algoritmo que evita esto se denomina Arranque Lento (Slow Start). Este opera observando que la tasa a la cual los nuevos paquetes deben ser inyectados en la red es la tasa a la cual los reconocimientos son retornados por el otro extremo.

El arranque lento como se mencionó antes añade otra ventana mas al emisor TCP: la ventana de congestionamiento denominada "cwnd". Cuando se establece una nueva conexión con un servidor en otra red, la ventana de congestión se inicializa al tamaño de un segmento (es decir, el tamaño de segmento anunciado por el otro extremo o por defecto, típicamente 536 o 512). A cada momento se recibe un acuse de recibo o reconocimiento, la ventana de congestión se incrementa por un segmento. El emisor pude transmitir hasta un mínimo de la ventana de congestión y la ventana advertencia. El control de flujo de la ventana de congestión es impuesto por el receptor. Lo anterior está basado en la evaluación del transmisor sobre la congestión de la red percibida; lo que sigue está relacionado al monto de espacio disponible del buffer en el receptor para ésta conexión.

El emisor comienza transmitiendo un segmento y esperando por su respectivo acuse de recibo. Cuando se recibe dicho reconocimiento, la ventana de congestión se incrementa de uno a dos, y dos segmentos pueden ser enviados. Cuando cada uno de esos dos segmentos son reconocidos, la ventana de congestión se incrementa a cuatro. Esto provee un crecimiento exponencial, aunque no es exactamente exponencial ya que el receptor puede retrasar sus reconocimientos, típicamente mandando un reconocimiento por cada dos segmentos que el recibe.

En algún punto la capacidad de la red puede ser alcanzada y entonces un enrutador intermedio comenzará a descartar paquetes. Esto le dice al transmisor que su ventana de congestión ha llegado a ser demasiado grande.

Algunas implementaciones primeras realizaban arranque lento (*slow start*) sólo si el otro extremo estaba en una red diferente. Las implementaciones actuales siempre realizan arranque lento.

1.3.4 Evasión de Tráfico:

A través del incremento exponencial de la tasa de envío causada por el algoritmo de arranque lento, la capacidad de la red será alcanzada en algunos momentos y un enrutador intermedio comenzará a descartar paquetes. Con el propósito de evitar esto, el algoritmo de arranque lento fue complementado a través del algoritmo de evasión de tráfico. Con éste algoritmo, la ventana de congestión se incrementa por cada reconocimiento como sigue:

$$cwnd + = 1/cwnd \qquad (1)$$

De esta manera la ventana de congestión se incrementa a lo sumo un segmento por cada viaje (viaje complementado de ida y vuelta en un enlace). La relación entre la evasión de tráfico y el arranque lento será explicado cuando se discuta el esquema de retransmisión usado en TCP.

El tráfico ó congestión puede suceder cuando los datos llegan de un tubo grande (de una LAN rápida) y quieren enviarse por un tubo pequeño (por una LAN lenta). La congestión puede también ocurrir cuando múltiples cadenas de entrada llegan hasta un enrutador cuya capacidad de salida es menor que la suma de las entradas. La evasión de tráfico es un camino para lidiar con los paquetes perdidos.

El algoritmo asume que la pérdida de los paquetes causados por daños es muy pequeña (mucho menos que el 1%), por tanto, la perdida de un paquete indica la congestión en algún lado de la red entre el emisor y el receptor. Hay dos indicadores de pérdida de paquete: la presencia de un receso y la aparición de acuses de recibo (ACK's) duplicados.

Los algoritmos de arranque lento y de evasión de tráfico son algoritmos independientes con objetivos distintos. Pero cuando se da la congestión el TCP debe reducir su tasa de transmisión de paquetes dentro de la red, y luego invocar el de

arranque lento para conseguir nuevamente el funcionamiento. En la práctica se implementan juntos.

La evasión de tráfico y arranque lento requieren que hayan dos variables que se mantienen para cada conexión: una ventana de congestión cwnd (*crowd window*) y una ssthresh (*slow start threshold size*). Los algoritmos combinados operan tal como sigue:

1. La inicialización para una conexión dada pone a cwnd al tamaño de un segmento y pone a ssthresh a 65535 bytes.

2. La rutina de salida TCP nunca manda mas del mínimo de cwnd mas la ventana de advertencia del receptor.

3. Cuando se da la congestión (indicada por el receso o por los ACKs duplicados), una mitad del tamaño de la ventana actual (el inínimo de la cwnd y la ventana de advertencia del receptor, pero al menos de dos segmentos) es guardada en ssthresh. Adicionalmente, si la congestión fue indicada por un receso, cwnd es puesto a un segmento (es decir, actúa el arranque lento).

Cuando se reconocen nuevos datos por el otro extremo, cwnd aumenta, pero la manera en que lo hace depende de cualquiera que sea la situación, ya sea que el TCP se encuentre realizando el arranque lento o que se encuentre realizando la evasión de tráfico (o evasión de congestión). Si la cwnd es menor ó igual que ssthresh, el TCP se encuentra en arranque lento; de otra forma, el TCP se encuentra realizando la evasión de tráfico. El arranque lento se mantiene hasta que el TCP está a la mitad de donde estaba cuando ocurrió la congestión (como el grabó la mitad del tamaño de la ventana que causó el problema en el paso 2), y luego entra en acción la evasión de tráfico. El arranque lento tiene a cwnd al porte de un segmento, y se es incrementado por un segmento con cada reconocimiento recibido. Como se mencionó anteriormente, esto abre la ventana exponencialmente: envía un segmento, luego dos, luego cuatro y así sucesivamente. La evasión de tráfico consigue que cwnd sea incrementada por un valor

igual a (segsize * segsize / cwnd) cada vez que se recibe un acuse de recibo o reconocimiento, donde segsize es el tamaño de segmento y cwnd se mantiene en bytes. Esto es un crecimiento lineal de la cwnd, comparada con el crecimiento exponencial que ocurre con el arranque lento. El incremento en cwnd debe ser a lo sumo un segmento cada tiempo de viaje redondo (sin tener en cuenta cuantos ACKs se han recibido en esta RTT[2]), en tanto que el arranque lento incrementa a cwnd por el número de reconocimientos recibidos en un tiempo de viaje redondo.

Muchas implementaciones añaden incorrectamente una pequeña fracción del tamaño de segmento (típicamente el tamaño del segmento dividido por 8) durante le evasión de tráfico. Esto es incorrecto y no debe ser estimulado en posteriores elaboraciones.

1.3.5 Retorno Exponencial y Estimación de Viaje Redondo:

Jacobson describió un método, en el que la estimación del tiempo de viaje redondo (RTT) se basa en el cálculo tanto de la media como de la varianza del RTT medido. Para simplificar el cálculo Jacobson sugirió que sólo la desviación media debió usarse en lugar de la desviación estándar. Esto lleva a las siguientes ecuaciones:

$$Err = M - A \quad (2)$$

$$A = A + gErr \quad (3)$$

$$D = D + h(|Err| - D) \quad (4)$$

$$RTO = A + 4D \quad (5)$$

[2] RTT (Round Trip Time) se le traduce como "Tiempo de Viaje Redondo" y se refiere al tiempo que se emplea para medir el ciclo de duración de viaje de un paquete, el cual consiste un envío mas un acuse de recibo.

Donde M es la RTT medida, A es la RTT suavizada, D es la desviación media suavizada y RTO es la actual estimación RTT. La ganancia g es para el promedio y se establece a 1/8, h es la ganancia para la desviación y se establece a ¼.

En las implementaciones originales del simulador TCP sólo un tiempo de viaje es medido por conexión a cada momento. Cuando una medición inicia, también se inicializa un contador con cero y el número de secuencia de paquete enviado se mantiene almacenado. El contador se incrementa cada 500 milisegundos. hasta que el reconocimiento del paquete enviado es recibido. Durante éste tiempo ninguna otra medición puede empezar. Después de recibir el reconocimiento para el paquete enviado, el valor del contador es empleado para actualizar el valor del RTO y se puede iniciar una nueva medición con el envío del próximo paquete. Por ello, sólo se puede hacer una medición por cada ventana enviada.

Si el temporizador se apaga antes de recibir el reconocimiento para ese paquete, éste tendrá que ser retransmitido y el valor del temporizador es doblado. Después de eso, el valor del temporizador es nuevamente doblado por cada retransmisión con un límite superior de 64 segundos. En la especificación original de éste algoritmo, el RTO fue calculado como A+2D. Sin embargo, éste mecanismo no era lo suficientemente flexible para contabilizar los cambios rápidos en el tiempo de viaje redondo, y ocasionalmente llevan a retransmisiones prematuras e innecesarias. En otras implementaciones TCP se usa un factor de 4 en lugar del 2 original para la varianza y casi todos las retransmisiones falsas desaparecieron.

La retransmisión pudo conducirnos al llamado *problema de ambigüedad de retransmisión*. Este problema radica en el hecho de que cuando llega un reconocimiento de paquete retransmitido no es posible determinar ya sea que ese reconocimiento corresponda a un paquete perdido ó al retransmitido. Como esto puede introducir un cierto error a la estimación del tiempo de viaje redondo, Karn[3] especifica que cuando el

[3] Ver ANEXO B.

reconocimiento para un paquete retransmitido llegue, los estimadores RTT no deben ser actualizados.

1.3.6 Retransmisión:

Hay dos formas para el emisor de detectar la pérdida de los paquetes:

a) *Ack's duplicados:* el receptor TCP puede detectar perdida de paquetes manteniendo la pista del número de secuencia en las cabeceras de los paquetes recibidos. Después de detectar un vacío en los números de secuencia, el receptor genera inmediatamente un reconocimiento, llamado *ack duplicado*, para el último byte correcto recibido. Esto se repite cada vez que un paquete fuera de orden se reciba.

b) *La expiración del temporizador de retransmisión:*
Después de haberse informado de la perdida, el transmisor tiene que retransmitir los paquetes perdidos y por tanto se tomarán los siguientes pasos:

1- Una variable llamada *ssthresh* se establece a la mitad de la cwnd actual.
2- La ventana de congestión es puesta al tamaño de un segmento.
3- El tamaño de la ventana de congestión es entonces incrementada por cada reconocimiento recibido. Mientras la cwnd es menor o igual al ssthresh el arranque lento actúa y la cwnd se incrementa por el tamaño de los segmentos reconocidos. De lo contrario, cwnd sólo se incrementa por 1/cwnd.

Dependiendo de la versión implementada, el emisor puede ahora enviar los datos ya sea comenzando por el número de byte que sigue al paquete perdido ó por donde se detuvo antes de la retransmisión (retransmisión rápida).

1.3.7 Retransmisión Rápida:

Modificaciones al algoritmo de evasión de tráfico fueron propuestas posteriormente. Antes de describir el cambio, hay que darse cuenta que el TCP puede generar un acuse de recibo inmediato (un ACK duplicado) cuando se recibe un segmento fuera de orden. Este ACK duplicado no debe ser retardado. El propósito de este ACK duplicado es permitir que el otro extremo sepa que se recibió un segmento fuera de orden, y decirle cual número de secuencia se esperaba.

Como el TCP no sabe si el ACK duplicado fue causado por un segmento perdido o por sólo un re ordenamiento de segmentos, el espera por un pequeño número de ACK's duplicados a ser recibidos. Se asume que si se trata de solo un re ordenamiento de segmentos, habrán solamente uno o dos ACK's duplicados antes de que el segmento re ordenado sea procesado, el cual generará entonces un nuevo ACK. Si tres o más ACK's duplicados son recibidos en una fila, es una fuerte indicación de que un segmento ha sido perdido. El TCP entonces realiza una retransmisión de lo que aparenta ser el segmento faltante, sin esperar que expire un temporizador de retransmisión

1.3.8 Recuperación Inmediata:

Luego de una rápida retransmisión se envía lo que parece ser el segmento faltante, evasión de tráfico, pero no se desarrolla el arranque lento. Esto es el algoritmo de recuperación inmediata. Es una mejoría que lleva a altas respuestas bajo congestiones moderadas, especialmente para grandes ventanas.

El motivo para no realizar el arranque lento en este caso es por que al recibir los ACK's duplicados le dice al TCP algo más que la simple perdida de un paquete. Como el receptor solo puede generar los ACK's duplicados cuando otro segmento se es recibido, ese segmento ha dejado la red y está ubicado en el buffer del receptor. Esto es,

aún hay datos fluyendo entre los dos extremos, y el TCP no quiere reducir el flujo abruptamente por aplicación de arranque lento.

Los algoritmos de retransmisión rápida y recuperación inmediata son usualmente implementados juntos.

1.3.9 Reconocimientos Retardados:

Para reducir el monto de los datos enviados, se retarda el envío de los reconocimientos hasta que el otro paquete es recibido ó hasta que un temporizador de reconocimiento se desactive.

1.4 CONCLUSION.

Como hemos apreciado, el desempeño del TPC incluye muchos servicios provistos por este protocolo que se combinan para hacer posible el transporte adecuado de los paquetes con información desde el transmisor hasta el destino. Los algoritmos de control de tráfico y los algoritmos de retransmisión de paquetes perdidos, han sido incluidos dentro del sistema de simulación aquí expuesto con el fin de poder seguir de una manera bastante fiel los servicios del control de transporte.

De la misma manera se han destacado ciertas simplificaciones de trabajo dentro del TCP con el afán de enfatizar solo las características mas importantes del mismo, pues el estudio principal consiste en demostrar los mecanismos empleados por el protocolo para transportar los paquetes con la información.

Finalmente, se incluye alguna información adicional de cómo se realizan los métodos en la retransmisión de paquetes, de la recuperación del sistema luego de una saturación, de las técnicas ideadas para evitar el congestionamiento a través del sistema, etc, y con ello poder

tener una apreciación mas amplia de lo que puede suceder a través de un sistema entre dos terminales comunicándose. En realidad éste proyecto no representa un sistema de comunicación propiamente dicho, pues no se pretende comunicar información específica, sino mas bien, el simulador tiene por objetivo hacer demostraciones de cómo se transportan los datos por medio de un protocolo que utiliza dentro de su mecanismo de operación la técnica de envío de paquetes con un acuse de recibo respectivo, y la forma de control sobre la pérdida de los mismos a través de un canal de comunicación.

CAPITULO 2

"El Protocolo de Control de Transporte TCP"

(Servicio de Trasporte de Flujo Confiable)

2.1 INTRODUCCIÓN.

Aunque el Protocolo de Control de Transmisión (TCP) se presenta como parte del grupo de protocolos Internet TCP/IP, en realidad es un protocolo independiente de propósitos generales que se puede adaptar para utilizarlo con otros sistemas de entrega. Por ejemplo, debido a que el TCP asume muy poco sobre la red adyacente, es posible usarlo en una sola red como Ethernet, así como una red de redes compleja.

2.2 CARACTERÍSTICAS DEL SERVICIO DE ENTREGA CONFIABLE.

La interfaz entre los programas de aplicación y el servicio TCP/IP de entrega confiable se puede caracterizar por cinco funciones:

- *Orientación de Flujo:* Cuando dos programas de aplicación (procesos de usuario) transfieren grandes volúmenes de datos, pensamos en los datos como un *flujo* de bits, divididos en *octetos* de 8 bits, que informalmente se conocen como *bytes*. El servicio de entrega de flujo en la maquina de destino pasa al receptor exactamente la misma secuencia de octetos que le pasa el transmisor en la maquina de origen.
- *Conexión de Circuito Virtual:* La transferencia de flujo es análoga a realizar una llamada telefónica. Antes de poder empezar la transferencia, los programas de aplicación, transmisor y receptor interactúan con sus respectivos sistemas operativos, informándose de la necesidad de realizar una transferencia de flujo.

Conceptualmente, una aplicación realiza una "llamada" que la otra tiene que aceptar. Los módulos de software de protocolo en los dos sistemas operativos se comunican al enviarse mensajes a través de una red de redes, verificando que la transferencia este autorizada y que los dos extremos estén listos. Una vez que se establecen todos los detalles, los módulos de protocolo informan a los programas de aplicación que se estableció una *conexión* y que la transferencia puede comenzar. Durante la transferencia, el software de protocolo en las dos maquinas continúa comunicándose para verificar que los datos se reciban correctamente. Si la comunicación no se logra por cualquier motivo (por ejemplo debido a que falle el hardware de red a lo largo del camino entre las maquinas), ambas maquinas detectarán la falla y la reportarán a los programas apropiados de aplicación. Se utiliza en término *circuito virtual* para describir dichas conexiones porque aunque los programas de aplicación visualizan la conexión como un circuito dedicado de hardware, la confiabilidad que se proporciona depende del servicio de entrega de flujo.

- *Transferencia con Memoria Interna:* Los programas de aplicación envían un flujo de datos a través del circuito virtual pasando repetidamente octetos de datos al software de protocolo. Cuando trasfieren datos, cada aplicación utiliza piezas del tamaño que encuentre adecuado, que pueden ser tan pequeñas cono un octeto. En el extremo receptor, el software de protocolo entrega octetos del flujo de datos en el mismo orden en que se enviaron, poniéndolos a disposición del programa de aplicación receptor tan pronto como se reciben y verifican. El software de protocolo puede dividir el flujo de paquetes, independientemente de las piezas que transfiera el programa de aplicación. Para hacer eficiente la transferencia de datos y minimizar el tráfico de red, las implantaciones por lo general recolectan datos suficientes de un flujo para llenar un datagrama razonablemente largo antes de transmitirlo a través de una red de redes. Por lo tanto, inclusive si el programa de aplicación genera en flujo un octeto a la vez, la transferencia a través de una red de redes puede ser sumamente eficiente. De forma similar, si el programa de aplicación genera bloques de datos muy largos, el software de protocolo puede dividir cada bloque en partes mas pequeñas para su transmisión. Para aplicaciones en las que los datos se deben

entregar aunque no se llene una memoria intermedia, el servicio de flujo proporciona un mecanismo de *empuje (push)* que las aplicaciones usan para forzar una transferencia. En el extremo transmisor, un empuje obliga al software de protocolo a transferir todos los datos generados sin tener que esperar a que se llene una memoria intermedia. Cuando llega al extremo receptor, el empuje hace que el TCP ponga los datos a disposición de la aplicación sin demora. Sin embargo, la función de empuje sólo garantiza que los datos se transferirán; no proporciona fronteras de registro. Por lo tanto, aun cuando la entrega es forzada, el software de protocolo puede dividir el flujo en formas inesperadas.

- *Flujo no Estructurado:* Es importante entender que el servicio de flujo TCP/IP no esta obligado a formar flujos estructurados de datos. Por ejemplo, no existe forma para que una aplicación de nómina haga que un servicio de flujo marque frontera entre los registros del empleado o que identifique el contenido del flujo como datos de nómina. Los programas de aplicación que utilizan el servicio de flujo deben entender el contenido del flujo y ponerse de acuerdo sobre su formato antes de iniciar una conexión.
- *Conexión Full Duplex:* Las conexiones proporcionadas por el servicio de flujo TCP/IP permiten la transferencia concurrentes en ambas direcciones. Dichas conexiones se conocen como *full duplex*. Desde el punto de vista de un proceso de aplicación, una conexión full duplex consiste en dos flujos independientes que se mueven en direcciones opuestas, sin ninguna interacción aparente. El servicio de flujo permite que un proceso de aplicación termine el flujo en una dirección mientras los datos continúen moviéndose en la otra dirección, haciendo que la conexión sea *half duplex*. La ventaja de una conexión full duplex es que el software subyacente de protocolo puede enviar en datagramas información de control de flujo al origen, llevando datos en la dirección opuesta. Este procedimiento de carga, transporte y descarga reduce el tráfico en la red.

2.3 PROPORCIONANDO CONFIABILIDAD.

El servicio de entrega de flujo confiable garantiza la entrega de los datos enviados de una maquina a otra sin pérdida o duplicación. Surge la pregunta: "¿cómo puede el software de protocolo proporcionar una trasferencia confiable si el sistema subyacente de comunicación sólo ofrece una entrega no confiable de paquetes?". La respuesta es complicada, pero la mayor parte de los protocolos confiables utilizan una técnica fundamental conocida como *acuse de recibo positivo con retransmisión.* La técnica requiere que un receptor se comunique con el origen y le envíe un mensaje de *acuse de recibo (ACK)* con forme recibe los datos. El transmisor guarda un registro de cada paquete que envía y espera un acuse de recibo antes de enviar el siguiente paquete. El transmisor también arranca un temporizador cuando envía un paquete y lo *retransmite* si dicho temporizador expira antes de que llegue un acuse de recibo.

En la figura 2.1 se muestra como transfiere datos el protocolo mas sencillo de acuse de recibo positivo.

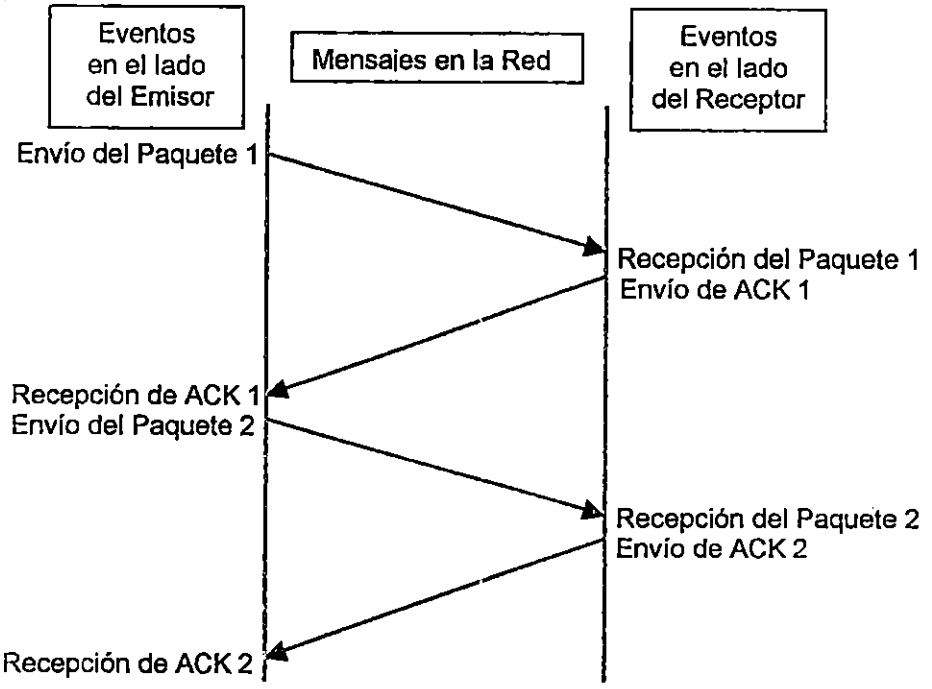


Figura 2.1. *Un protocolo que se vale de reconocimientos o acuses de recibo positivos, con retransmisión, en la cual el emisor espera un acuse de recibo para cada paquete enviado. La distancia vertical bajo la figura representa el incremento en el tiempo y la líneas que cruzan en diagonal representan la transmisión de paquetes de red.*

En la figura 2.1, los eventos en el transmisor y receptor se muestran a la izquierda y derecha, respectivamente. Cada línea diagonal que cruza por el centro muestra la transferencia de un mensaje a través de la red.

En la figura 2.2, se utiliza el mismo diagrama de formato que en la figura 2.1 para mostrar qué sucede cuando se pierde o corrompe un paquete. El transmisor arranca un temporizador después de enviar un paquete. Cuando termina el tiempo, el transmisor asume que el paquete se perdió y lo vuelve a enviar.

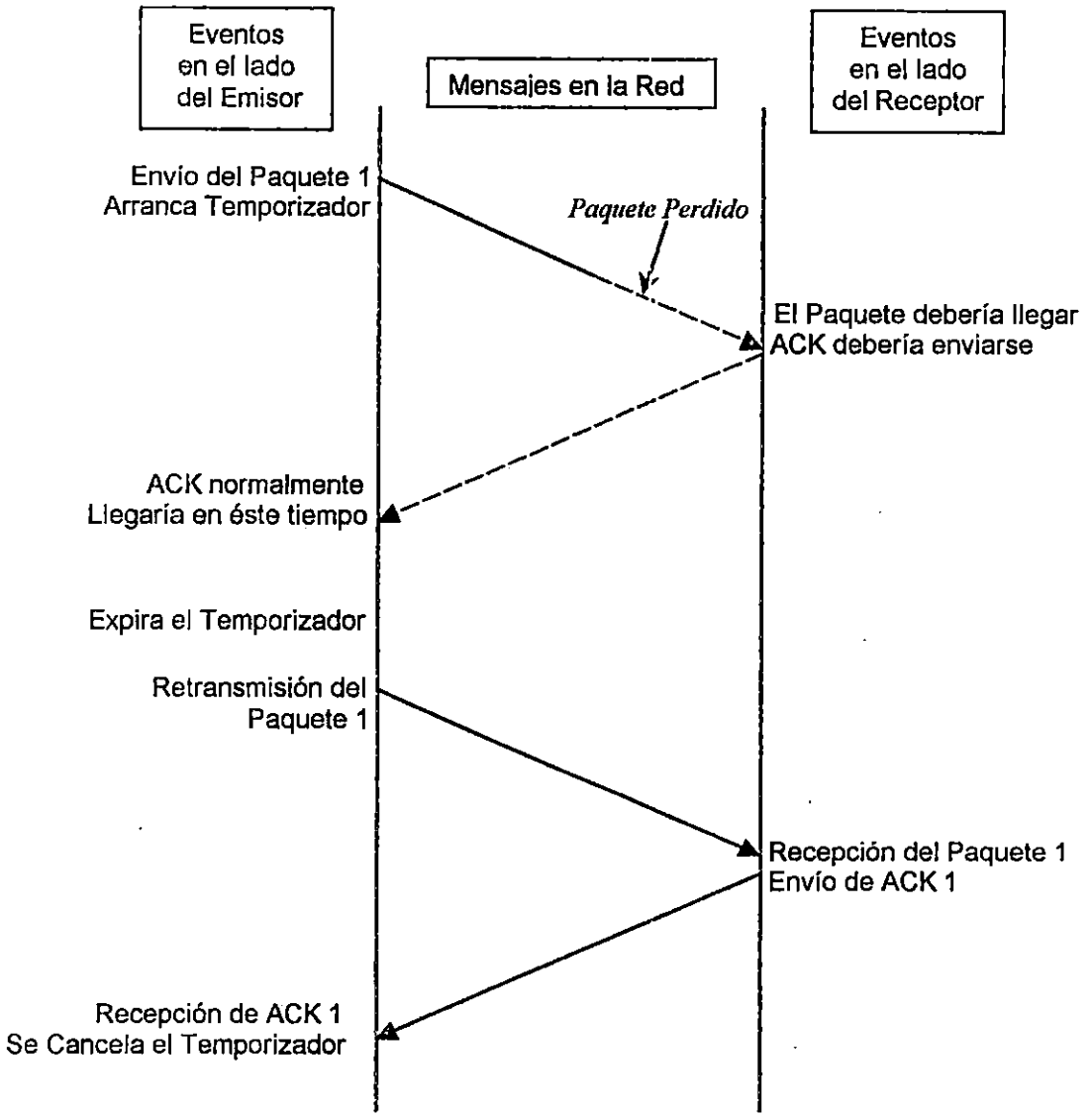


Figura 2.2. *Tiempo excedido y retransmisión que ocurre cuando un paquete se pierde. La línea punteada muestra el tiempo que podría ocuparse para la transmisión de un paquete y su acuse de recibo, si no se perdiera el paquete.*

El problema final de confiabilidad surge cuando un sistema subyacente de entrega de paquetes los duplica. Los duplicados también pueden surgir cuando las redes tienen grandes retrasos que provocan la retransmisión prematura. La solución de la duplicación requiere acciones cuidadosas ya que tanto los paquetes como los acuses de recibo se pueden duplicar. Por lo general, los protocolos confiables detectan los paquetes duplicados al asignar a cada uno un número de secuencia y al obligar al receptor a recordar qué número de secuencia recibe. Para evitar la confusión causada por acuses de recibo retrasados o duplicados, los protocolos de acuses de recibo positivo envían los números de secuencia dentro de los acuses, para que el receptor pueda asociar correctamente los acuses de recibo con los paquetes.

La *ventana deslizante*, hace que la transmisión de flujo sea eficiente. Para entender lo que motiva a usar ventanas deslizantes hay que recordar la secuencia de eventos que se muestran en la figura 2.1. A fin de lograr la confiabilidad, el transmisor envía un paquete y espera un acuse de recibo antes de enviar otro. Como se muestra en la figura 2.1, los datos sólo fluyen entre las maquinas en una dirección a la vez, inclusive si la red tiene capacidad para comunicación simultánea en ambas direcciones. La red estará del todo ociosa durante el tiempo en que las maquinas retrasen sus respuestas (por ejemplo, mientras las maquinas computan rutas o sumas de verificación). Si imaginamos una red con altos retrasos en la transmisión, el problema es evidente: "*Un protocolo simple de acuses de recibo positivos ocupa una cantidad sustancial de ancho de banda de red debido a que debe retrasar el envío de un nuevo paquete hasta que reciba un acuse de recibo del paquete anterior*".

La técnica de ventana deslizante es una forma mas compleja de acuse de recibo positivo y retransmisión que el sencillo método mencionado antes. Los protocolos de ventana deslizante ocupan el ancho de banda de red de mejor forma, ya que permiten que el transmisor envíe varios paquetes sin esperar un acuse de recibo. La mejor manera de visualizar la operación de ventana deslizante, es pensar en una secuencia de paquetes que se transmitirán como se muestra en la figura 2.3. El protocolo coloca una *ventana* pequeña y de tamaño fijo en la secuencia, y transmite todos los paquetes que residan dentro de la ventana.

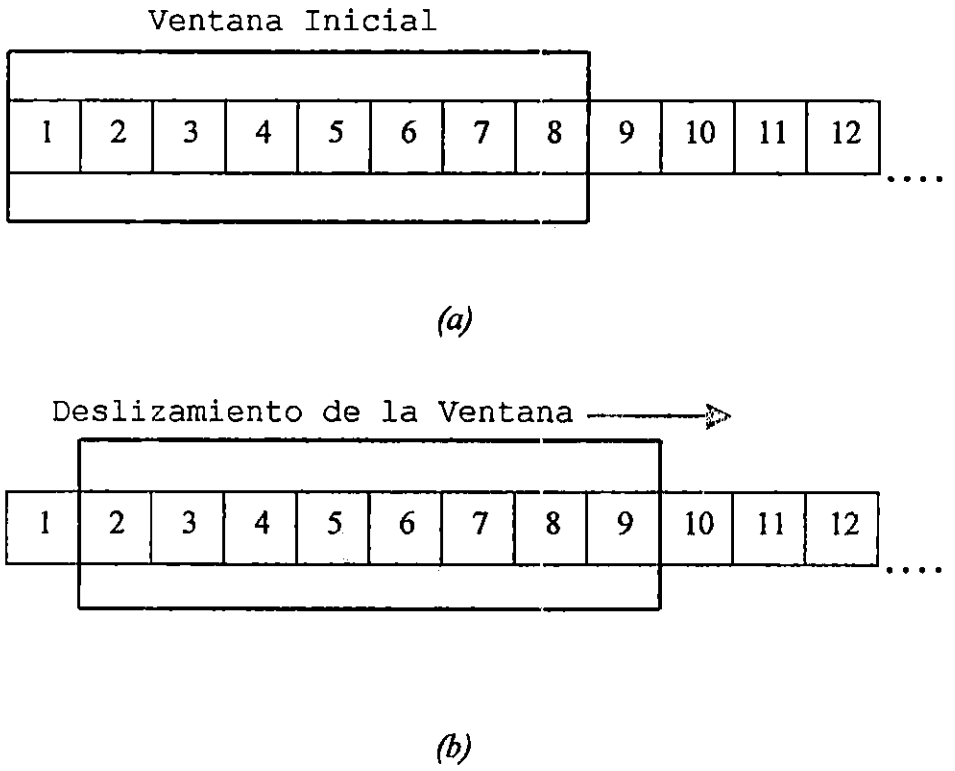


Figura 2.3. *(a) Un protocolo de ventana deslizante con ocho paquetes en la ventana, y (b) La ventana que se desliza hacia el paquete 9 puede enviarse cuando se recibe un acuse de recibo del paquete 1. Únicamente se re retransmiten los paquetes sin acuse de recibo.*

Se dice que un paquete es *unacknowledged* o sin acuse de recibo[4] si se transmitió pero no se recibió ningún acuse de recibo. Técnicamente, el número de paquetes sin acuse de recibo en un tiempo determinado depende del *tamaño de la ventana* y está limitado a un número pequeño y fijo. Por ejemplo, en un protocolo de ventana deslizante con un tamaño de ventana de 8, se permite el transmisor enviar ocho paquetes antes de recibir un acuse de recibo.

Como se muestra en la figura 2.3, una vez que el transmisor recibe un acuse de recibo para el primer paquete dentro de la ventana, "mueve" la misma y envía el siguiente paquete. La ventana continuará moviéndose en tanto se reciban acuses de recibo.

[4] No confirmado o paquete del cual no se recibió acuse de recibo.

El desempeño de los protocolos de ventana deslizante depende del tamaño de la ventana y de la velocidad en que la red acepta paquetes. En la figura 2.4, se muestra un ejemplo de la operación de un protocolo de ventana deslizante cuando se envían tres paquetes. Nótese que el transmisor los envía antes de recibir cualquier acuse de recibo.

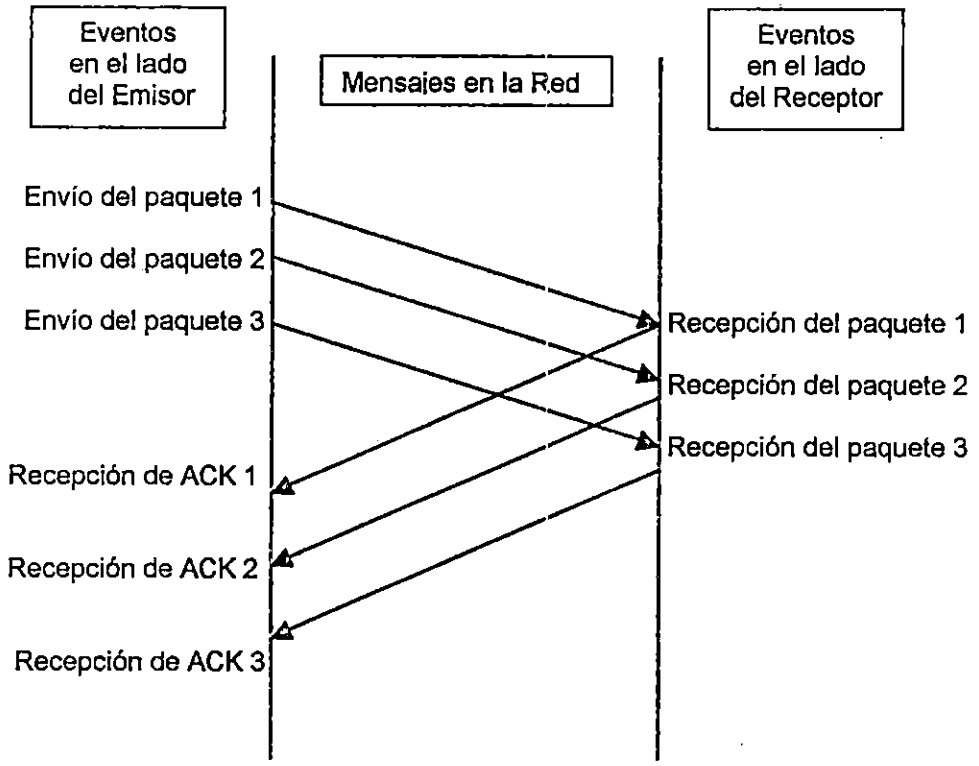


Figura 2.4. *Ejemplo de tres paquetes transmitidos mediante un protocolo de ventana deslizante. El concepto clave es que el emisor puede transmitir todos los paquetes de la ventana sin esperar un acuse de recibo.*

Con un tamaño de ventana de 1, un protocolo de ventana deslizante sería idéntico a un protocolo simple de acuse de recibo positivo. Al aumentar el tamaño de la ventana, es posible eliminar completamente el tiempo ocioso de la red. Esto es, en una situación estable, el transmisor puede enviar paquetes tan rápido como la red los pueda transferir. El punto principal es que: "*Como un protocolo de ventana deslizante bien establecido mantiene la red completamente saturada de paquetes, con él se obtiene una generación de salida substancialmente mas alta que con un protocolo simple de acuse de recibo positivo*".

Conceptualmente, un protocolo de ventana deslizante siempre recuerda qué paquetes tienen acuse de recibo y mantiene un temporizador separado para cada paquete sin acuse de recibo.

2.4 PROTOCOLO DE CONTROL DE TRANSPORTE.

Ya que se explicó el principio de las ventanas deslizantes, podemos examinar el servicio de flujo confiable proporcionado por el grupo de protocolos TCP/IP de Internet. El servicio lo define el *Protocolo de Control de Transporte* o TCP. El servicio de flujo confiable es tan importante que todo el grupo de protocolos se conoce como TCP/IP y es importante entender que: "*El TCP es un protocolo de comunicación, no una pieza de software*". La diferencia entre un protocolo y el software que lo implementa es análoga a la diferencia entre la definición de un lenguaje de programación y un compilador.

¿Qué proporciona el TCP exactamente? El TCP especifica el formato de datos y los acuses de recibo que intercambian dos computadoras para lograr una transferencia confiable, así como los procedimientos que la computadora usa para asegurarse de que los datos lleguen de manera correcta. También especifica como el software TCP distingue el correcto entre muchos distintos en una misma maquina, y como las maquinas en comunicación resuelven errores como la perdida o duplicación de paquetes. El protocolo también especifica como dos computadoras inician una transferencia de flujo TCP y como se ponen de acuerdo cuando se completa.

2.5 SEGMENTOS, FLUJOS Y NUMEROS DE SECUENCIA.

El TCP visualiza el flujo de datos como una secuencia de octetos (o bytes) que divide en *segmentos* para su transmisión. Por lo general, cada segmento viaja a través de una red de redes como un solo datagrama IP.

El TCP utiliza un mecanismo especializado de ventana deslizante para solucionar dos problemas importantes: la transmisión eficiente y el control de flujo. Al igual que el protocolo de ventana deslizante descrito anteriormente, el mecanismo de ventana del TCP hace posible enviar varios segmentos antes de que llegue un acuse de recibo. Hacerlo así aumenta la generación total de salida ya que mantiene ocupada la red. La forma TCP de un protocolo de ventana deslizante también soluciona el problema de *control de flujo* de extremo a extremo, al permitir que el receptor restrinja la transmisión hasta que tenga espacio suficiente en memoria intermedia para incorporar mas datos.

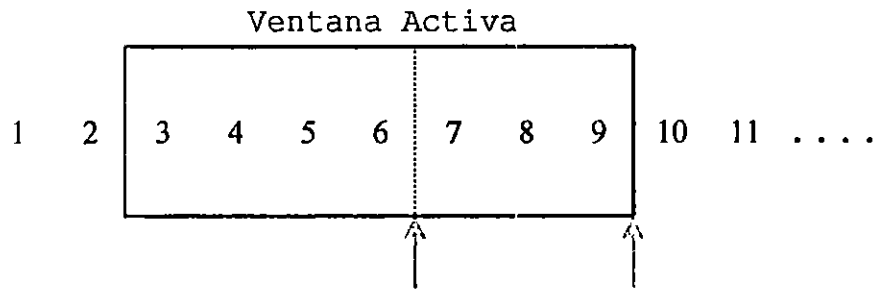


Figura 2.5. *Ejemplo de la ventana deslizante del TCP. Los octetos hasta el dos se han enviado y reconocido, los octetos del 3 al 6 han sido enviados pero no reconocidos, los octetos del 7 al 9 no se han enviado pero serán enviado sin retardo y los octetos del 10 en adelante no pueden ser enviados hasta que la ventana se mueva.*

El mecanismo TCP opera a nivel de octeto, no a nivel de segmento ni de paquete. Los octetos del flujo de datos se numeran de manera secuencial, y el transmisor guarda tres valores asociados con cada conexión. Los apuntadores definen una ventana deslizante, como se muestra en la figura 2.5. El primer apuntador marca el extremo izquierdo de la ventana deslizante, separa los octetos que ya se enviaron y envía el acuse de recibo de los octetos ya enviados. Un segundo apuntador marca el extremo derecho de la ventana deslizante y define el octeto mas alto en la secuencia que se puede enviar antes de recibir mas acuses de recibo. El tercer apuntador señala la frontera dentro de la ventana que separa los octetos que ya se enviaron de los que todavía no se envían. El software de protocolo envía sin retraso todos los octetos dentro de la ventana, por lo que en general, la frontera dentro de la ventana se mueve rápidamente de izquierda a derecha.

2.6 TAMAÑO VARIABLE DE VENTANA Y CONTROL DE FLUJO.

Una diferencia entre el protocolo TCP de ventana deslizante y el protocolo simplificado de ventana deslizante, presentado anteriormente, es que el TCP permite que el tamaño de la ventana varíe. Cada acuse de recibo, que informa cuantos octetos se recibieron, contiene un *aviso de ventana*, que especifica cuantos octetos adicionales de datos está preparado para aceptar el receptor. Pensemos en el aviso de ventana como la especificación del tamaño actual de la memoria intermedia del receptor. En respuesta a un aumento en el aviso de ventana, el transmisor aumenta el tamaño de su ventana deslizante y procede al envío de octetos de los que todavía no se tiene un acuse de recibo. En respuesta a una disminución en el aviso de ventana, el transmisor disminuye el tamaño de su ventana y deja de enviar los octetos que se encuentran mas allá de la frontera. El software TCP no debe contradecir los anuncios previos, reduciendo la posición aceptable de la ventana, que pasó anteriormente, en flujo de octetos. De hecho, los anuncios mas pequeños acompañan a los acuses de recibo, así que el tamaño de la ventana cambia en el momento que se mueve hacia adelante.

La ventaja de utilizar una ventana de tamaño variable es que ésta proporciona control de flujo así como una transferencia confiable. Si la memoria intermedia del receptor se llena, no puede aceptar mas paquetes, así que envía un anuncio de ventana mas pequeño. En caso extremo, el receptor anuncia un tamaño de ventana igual a cero para detener toda la transmisión. Después, cuando hay memoria intermedia disponible, el receptor anuncia un tamaño de ventana distinto a cero para activar de nuevo el flujo de datos[5].

La sobrecarga de las maquinas intermedias se conoce como *congestionamiento* y los mecanismos que resuelven el problema se conocen como mecanismos o algoritmos de *control de congestionamiento*. El TCP emplea su esquema de ventana deslizante para resolver el problema de control de flujo extremo a extremo; no cuenta con un mecanismo explicito para el control de congestionamientos.

---

[5] Hay dos excepciones para la transmisión cuando el tamaño de la ventana es cero. Primero, cuando un emisor esta autorizado a transmitir un segmento con el bit de urgente activo para informar al receptor que está disponible un dato urgente. Segundo, para evitar un fin de cronómetrado potencial si un anuncio diferente de cero se pierde luego de que el tamaño de la ventana llega a cero, el emisor prueba una ventana de tamaño cero periódicamente.

2.7 FORMATO DEL SEGMENTO TCP.

La unidad de transferencia entre el software TCP de dos maquinas se conoce como *segmento*. Los segmentos se intercambian para establecer conexiones, transferir datos, enviar acuses de recibo, anunciar los tamaños de ventana y para cerrar conexiones. Debido a que el TCP usa acuses de recibo incorporados, un acuse que viaja de la maquina A a la maquina B puede viajar en el mismo segmento en el que viajan los datos de la maquina A a la maquina B, aun cuando el acuse de recibo se refiera a los datos enviados de B hacia A[6]. En la figura 2.6 se muestra el formato del segmento TCP.

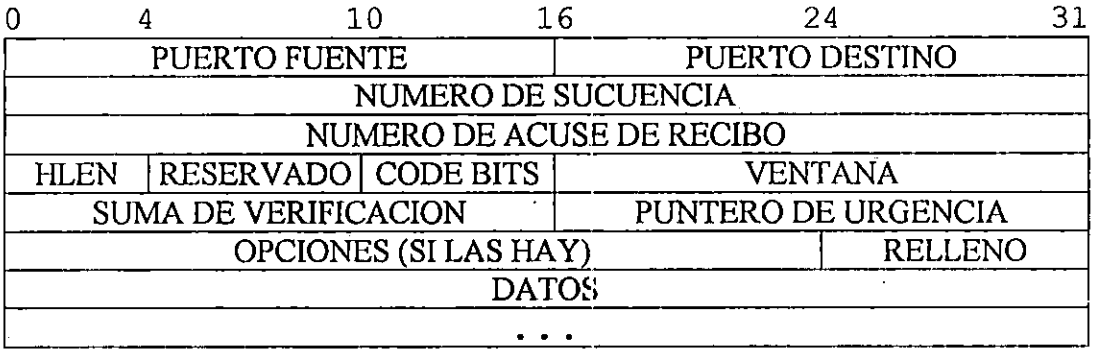


Figura 2.6. *Formato de un segmento TCP con un encabezado TCP seguido de datos. Los segmentos se utilizan para establecer conexiones, así como para transportar datos y acuses de recibo.*

Cada segmento se divide en dos partes: encabezado y datos. El encabezado, conocido como *encabezado TCP*, transporta la identificación y la información de control. Los campos PUERTO FUENTE Y DESTINO contienen los números de puerto TCP que identifican a los programas de aplicación en los extremos de la conexión. El campo NUMERO DE SECUENCIA identifica la posición de los datos del segmento en el flujo de datos del transmisor. El campo NUMERO DE ACUSE DE RECIBO identifica el numero de octetos que la fuente espera recibir después. Observe que el numero de secuencia se

[6] En la practica este tipo de incorporación no se presenta con frecuencia ya que la mayor parte de las aplicaciones no envía datos en ambas direcciones de manera simultánea.

refiere al flujo que va en la misma dirección que el segmento, mientras que el numero de acuse de recibo se refiere al flujo que va en la dirección opuesta en el segmento.

El campo HLEN[7] contiene un numero entero que especifica la longitud del encabezado, del segmento, medida en múltiplos de 32 bits. Es necesario porque el campo OPCIONES varía en su longitud, dependiendo en qué opciones se haya incluido. Así el tamaño del encabezado TCP varía dependiendo de las opciones seleccionadas. El campo de 6 bits, etiquetado como CODE BITS, es para determinar el propósito y contenido del segmento. El software TCP informa sobre cuantos datos esta dispuesto a aceptar cada vez que envía un segmento, al especificar su tamaño de memoria intermedia en el campo VENTANA. El campo contiene un numero entero sin signo de 16 bits en el orden de octetos estándar de red. Los anuncios de ventana proporcionan otro ejemplo de acuse de recibo de carga, transporte y descarga ya que acompañan a todos los segmentos, tanto a los que llevan datos, como a los que sólo llevan un acuse de recibo. Los detalles de cómo el TCP informa al programa de aplicación sobre datos urgentes dependen del sistema operativo de la maquina. El mecanismo usado para marcar los datos urgentes cuando se transmiten en un segmento consiste en un bit de código URG y en un campo PUNTERO DE URGENCIA. Cuando se activa el bit URG, el indicador urgente especifica la posición dentro del segmento en la que terminan los datos urgentes.

2.8 OPCION DE TAMAÑO MÁXIMO DE SEGMENTO.

No todos los segmentos que se envían a través de una conexión serán del mismo tamaño. Sin embargo, ambos extremos necesitan acordar el tamaño máximo de los segmentos que transferirán. El software TCP utiliza el campo OPCIONES para negociar con el software TCP en el otro extremo de la conexión; una opción permite que el software TCP especifique el *Tamaño Máximo de Segmento (MSS)* que esta dispuesto a recibir. Por ejemplo cuando un sistema incorporado que solamente contiene unos cantos cientos de

[7] La especificación indica que el campo *HLEN* es el desplazamiento del área de datos dentro del segmento.

octetos de memoria intermedia se conecta con una gran supercomputadora, puede negociar un MSS que restrinja los segmentos para que quepan en la memoria intermedia.

2.9 COMPUTO DE LA SUMA DE VERIFICACIÓN.

El campo SUMA DE VERIFICACIÓN (CHECKSUM) en el encabezado TCP contiene una suma de verificación de números enteros y 16 bits que se utiliza para verificar la integridad de los datos así como del encabezado TCP. Para computar la suma de verificación, el software TCP en la maquina transmisora coloca un *seudo-encabezado* en el segmento, agrega suficientes bits en cero para lograr que el segmento sea un múltiplo de 16 bits y calcula la suma de 16 bits sobre todo el resultado. El TCP no cuenta todo el seudo-encabezado ni los caracteres de relleno en la longitud del segmento, ni tampoco los transmite. También asume que el campo de suma de verificación por sí mismo es de cero, para propósitos de la suma. Como en el caso de otras sumas de verificación, el TCP usa aritmética de 16 bits y toma el complemento a uno del complemento a uno de la suma. En la localidad receptora, el software TCP realiza el mismo cómputo para verificar que el segmento llega intacto. En la figura 2.7, se muestra el formato del seudo-encabezado empleado en el cómputo de la suma de verificación.

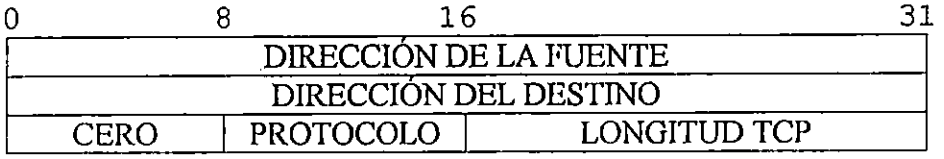


Figura 2.7. *Formato del seudo-encabezado utilizado en el cálculo de la suma de verificación del TCP.*

El TCP transmisor asigna al campo PROTOCOLO el valor que utilizará el sistema subyacente de entrega en su campo de tipo de protocolo. Para los datagramas IP que transporten TCP, el valor es 6. El campo LONGITUD TCP especifica la longitud total del segmento TCP, incluyendo su encabezado. En el extremo receptor, la información utilizada

en el seudo-encabezado se extrae del datagrama IP que transportó el segmento y se incluye en el cómputo de la suma para verificar que el segmento llegó intacto al destino correcto.

2.10 ACUSES DE RECIBO Y RETRANSMISIÓN.

Como el TCP envía los datos en segmentos de longitud variable, y debido a que los segmentos retransmitidos pueden incluir mas datos que los originales, los acuses de recibo no pueden remitirse fácilmente a los datagramas o segmentos. De hecho, se remiten a una posición en el flujo, utilizando los números de secuencia de flujo. El receptor recolecta octetos de datos de los segmentos entrantes y reconstruye una copia exacta del flujo que se envía. Como los segmentos viajan en datagramas IP, se pueden perder o llegar en desorden; el receptor usa los números de secuencia para reordenar los segmentos. En cualquier momento, el receptor tendrá cero o mas octetos reconstruidos contiguamente desde el comienzo del flujo, pero puede tener piezas adicionales del flujo de datagramas que hayan llegado en desorden. El receptor siempre acusa recibo del prefijo contiguo mas largo del flujo que se recibió correctamente. Cada acuse de recibo especifica un valor de secuencia mayor en una unidad, con respecto al octeto de la posición mas alta en el prefijo contiguo que recibió. Por lo tanto, el transmisor recibe una realimentación continua del receptor conforme progresa el flujo, por tanto "*Un acuse de recibo TCP especifica el número de secuencia del siguiente octeto que el receptor espera recibir*"

2.11 TIEMPO LIMITE Y RETRANSMISIÓN.

Una de las ideas mas importantes y complejas del TCP es parte de la forma en que maneja la terminación de tiempo (*time out*) y la retransmisión. Al igual que otros protocolos confiables, el TCP espera que el destino envíe acuses de recibo siempre que recibe exitosamente nuevos octetos del flujo de datos. Cada vez que envía un segmento, el TCP arranca un temporizador y espera un acuse de recibo. Si se termina el tiempo antes de

que se acusen de recibidos los datos en el segmento, el TCP asume que dicho segmento se perdió o corrompió y lo retransmite.

Para entender por que el algoritmo TCP de retransmisión difiere del algoritmo usado en muchos protocolos de red, necesitamos recordar que el TCP esta diseñado para emplearse en un ambiente de red de redes, por tanto un segmento que viaja entre dos maquinas puede atravesar una sola red de poco retraso (por ejemplo, una LAN de alta velocidad) o puede viajar a través de varias redes intermedias y de varios enrutadores. Por tanto es imposible saber con anticipación que tan rápido regresaran los acuses de recibo al origen. Además, el retraso en cada enrutador depende del tráfico, por lo que el tiempo total necesario para que un segmento viaje al destino y para que un acuse de recibo regrese al origen varía dramáticamente de un ejemplo a otro. En la figura 2.8, en la que se muestran medidas de tiempos de viaje redondo a través de la red Internet global para 100 paquetes consecutivos, se ilustra el problema. El software TCP debe incorporar las amplias diferencias en el tiempo necesario para llegar a varios destinos, así como los cambios en el tiempo necesario para llegar a cierto destino conforme varía la carga de tráfico.

El TCP maneja los retardos variables en la red de redes al utilizar un *algoritmo de retransmisión*. En esencia, el TCP monitora el desempeño de cada conexión y deduce valores razonables para la terminación de tiempo. Conforme cambia el desempeño de una conexión, el TCP revisa su valor de terminación de tiempo (por ejemplo, se adapta al cambio).

Para recolectar los datos necesarios para un algoritmo adaptable, el TCP registra la hora en la que se envía cada segmento y la hora en la que se recibe un acuse de recibo para los datos en el segmento. Considerando las dos horas, el TCP computa el tiempo transcurrido, conocido como *Tiempo ejemplo de viaje redondo* o *ejemplo de viaje redondo.* Siempre que obtiene un nuevo ejemplo de viaje redondo, el TCP ajusta su noción de tiempo de viaje redondo promedio para la conexión. Por lo general, el software TCP almacena el tiempo estimado de viaje redondo, RTT (*round trip time*), como promedio calculado y utiliza nuevos ejemplos de viaje redondo para cambiar lentamente dicho promedio. Por ejemplo,

para computar un nuevo cálculo de promedio, una técnica antigua para promediar se valía de un factor constante, α, donde $0 \leq \alpha \leq 1$, para calcular el promedio anterior contra el último ejemplo de viaje redondo:

$$\mathbf{RTT = (\alpha * Old\_RTT) + ((1-\alpha) * New\_Round\_Trip\_Sample)}$$

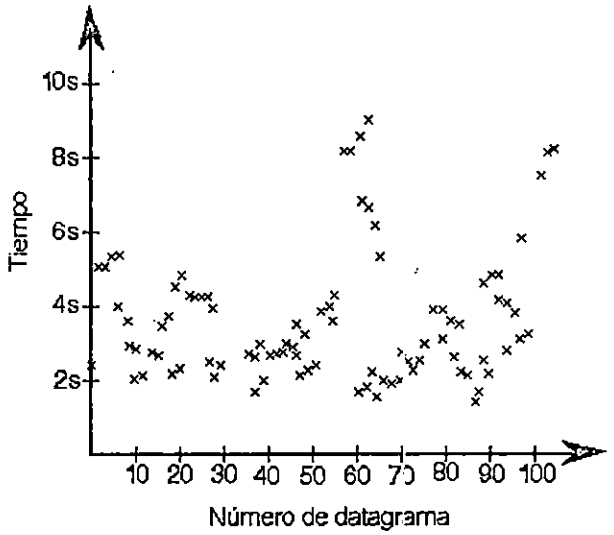


Figura 2.8. *Gráfico de tiempos de viaje redondo para 100 datagramas IP sucesivos. Aun cuando la mayor parte de Internet opera ahora con retardos mucho menores, los retardos varían aún en tiempo.*

Escoger un valor cercano a 1 para α, hace que el promedio calculado sea inalterable ante los cambios mínimos de tiempo (por ejemplo, un solo segmento que encuentra un gran retraso). Escoger un valor cercano a 0 para α, hace que el promedio calculado responda con rapidez a los cambios en el retraso.

Cuando envía un paquete, el TCP computa un valor de terminación de tiempo como una función de la estimación actual para viaje redondo. Las implementaciones antiguas del TCP

se valían de un factor constante de cálculo, β (β>1), y la terminación de tiempo era mayor que la estimación actual de viaje redondo:

Terminación de Tiempo = β * RTT

Escoger un valor para β puede ser difícil. Por una parte, para detectar con rapidez la perdida de paquetes, el valor de terminación de tiempo debe acercarse al tiempo actual de viaje redondo (por ejemplo, β debe acercarse a 1). La rápida detección de perdida de paquetes mejora la producción de salida porque el TCP no esperará un tiempo innecesariamente largo para retransmitir. Por otra parte, si β = 1, el TCP se vuelve muy ansioso y cualquier retraso pequeño causará una retransmisión innecesaria, que desperdiciará el ancho de banda de la red. La especificación original recomienda establecer β = 2, pero trabajos mas recientes, han producido mejores técnicas para el ajuste de la terminación de tiempo.

2.12 CONCLUSION

Podemos observar del TCP, un protocolo complejo que maneja las comunicaciones sobre una amplia variedad de tecnologías de red subyacentes. Mucha gente asume que, como el TCP aborda tareas mucho mas complejos que otros protocolos de transporte, el código debe ser incómodo e ineficaz. Sorpresivamente, en general lo que hemos analizado no parece entorpecer el desempeño del TCP. Experimentos realizados en Berkeley han demostrado que el mismo TCP que opera en forma eficiente en la red global de Internet puede proporcionar un desempeño sostenido a 8 Mbps con datos de usuario entre dos estaciones de trabajo en una red Ethernet a 10 Mbps.[8] Los investigadores de Cray Research, Inc. han demostrado que el desempeño del TCP se acerca a un gigabit por segundo.

[8] Encabezado Ethernet, IP y TCP con el espacio requerido de "Inter.-packet", representan en amplitud de banda remanente.

El Protocolo de Control de Transporte (TCP por sus siglas en Inglés) define un servicio clave proporcionando para una red de redes llamado *entrega de flujo confiable*. El TCP proporciona una conexión tipo full duplex entre dos máquinas, lo que les permite intercambiar grandes volúmenes de datos de manera eficaz.

Dado que utiliza un protocolo de ventana deslizante, el TCP puede hacer eficiente el uso de la red. Como se hacen pocas suposiciones sobre el sistema de entrega subyacente, el TCP es lo suficientemente flexible como para operar sobre una gran variedad de sistemas de entrega. Ya que proporciona un control de flujo, el TCP permite que el sistema cuente con una amplia variedad de velocidades para la comunicación.

La unidad básica de transferencia utilizada por el TCP es un segmento. Los segmentos se emplean para transferir datos o información de control (por ejemplo, para permitir que el software TCP en dos máquinas establezca o interrumpa la comunicación). El formato de los segmentos permite a una máquina incorporar un acuse de recibo para datos que fluyen en una dirección, incluyéndolos en el encabezado del segmento de datos que fluyen en el sentido opuesto.

El TCP implanta el control de flujo estableciendo, en el anuncio del receptor, la cantidad de datos que esta dispuesto a aceptar. También soportas mensajes fuera de banda utilizando una capacidad de datos urgentes y forzando la entrega por medio de un mecanismo de empuje.

El estándar TCP actual especifica un retroceso exponencial para los temporizadores de retransmisión y algoritmos de prevención de congestionamiento como el de arranque lento, disminución multiplicativa e incremento aditivo.

CAPITULO 3

"El Dominio de Eventos Discretos (DE) de Ptolemy"

3.1 INTRODUCCIÓN.

El dominio de los eventos discretos (DE) en Ptolemy provee un entorno general para las simulaciones orientadas en el tiempo de sistemas tales como redes con colas, redes de comunicaciones, y arquitectura de computadoras en alto nivel. En este dominio, cada partícula representa un *evento* que corresponde a un cambio dentro del estado del sistema. Los organizadores del DE procesan los eventos en orden cronológico. Como el intervalo de tiempo entre eventos generalmente no esta fijado, cada partícula tiene asociada una *marca de tiempo*[9]. Estas marcas de tiempo son generadas por los bloques que producen la partícula y que al mismo tiempo están basadas en la marca de tiempo de las partículas de entrada y la latencia del bloque.

3.2 UN VISTAZO A LAS ESTRELLAS DEL DOMINIO "DE" UTILIZADAS EN SIMULACION.

El modelo de computación en el dominio DE lo hace mas apropiado para modelar aquellos sistemas de alto nivel. Por esta razón, las estrellas en el dominio DE suelen ser mas complicadas, y mas especializadas que aquellas del dominio SDF (Dominio del Flujo de Datos Síncronos). En la figura 3.1 se presenta parte de la paleta de las estrellas de alto nivel en el dominio DE y que se han empleado para simular el protocolo TCP.

[9] Timestamp o marca de tiempo, se refiere a un número real que está asociado a una partícula dentro del dominio particular, que indican el punto en tiempo de simulación durante el cual una partícula es válida.

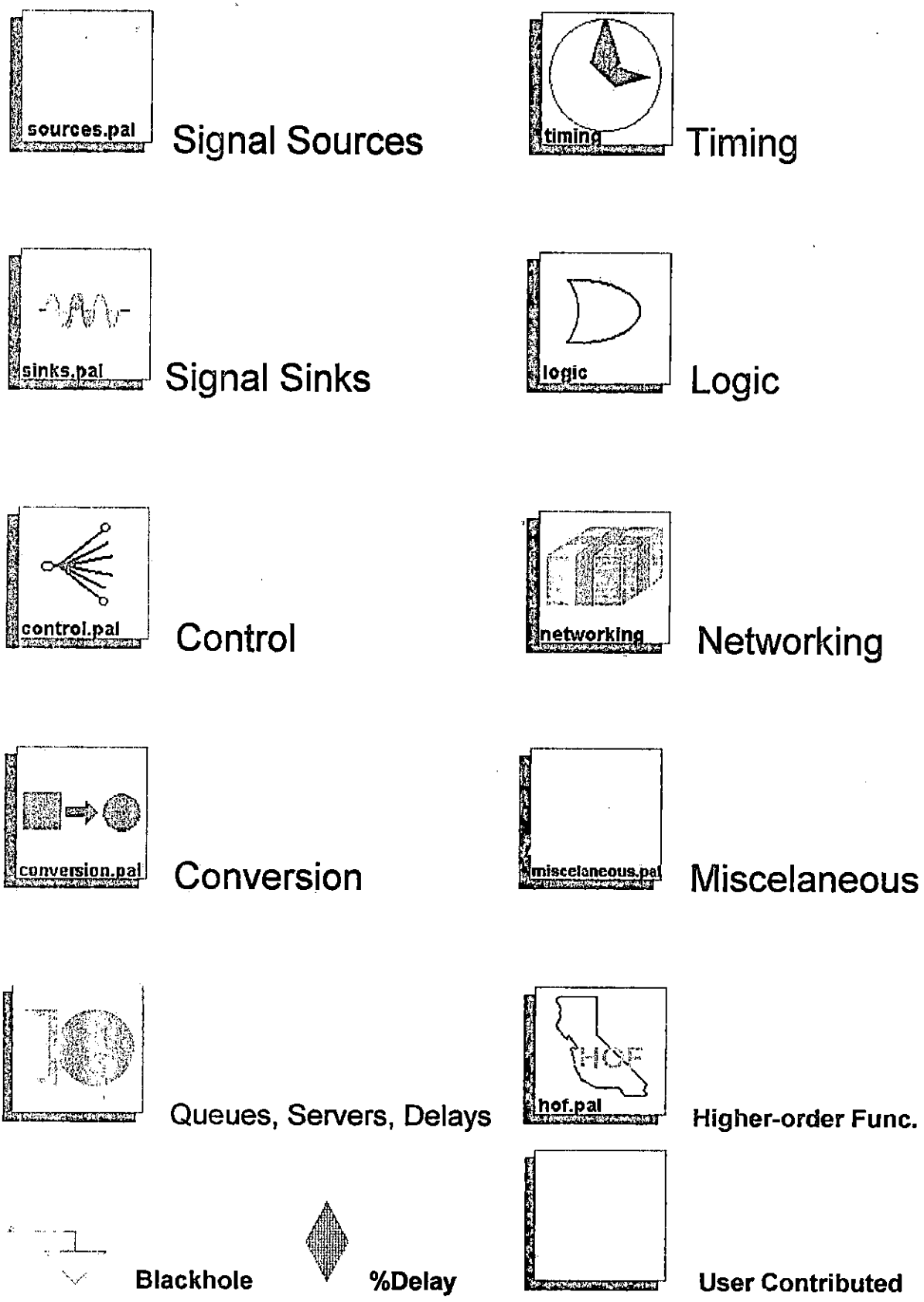
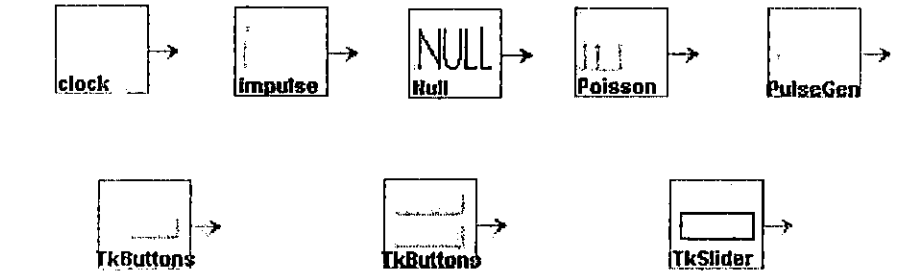


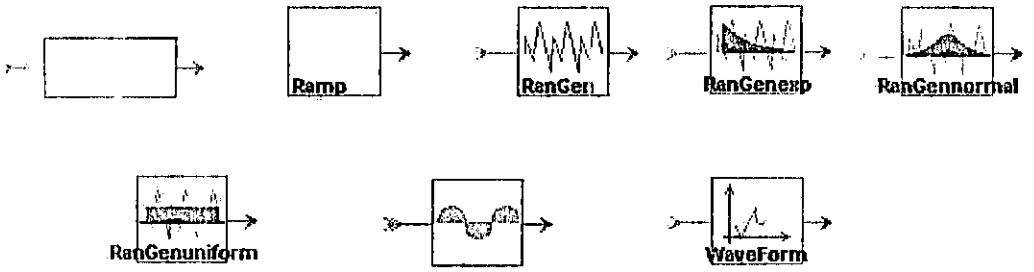
Figura 3.1. *Paleta de estrellas de alto nivel para eventos discretos usados en el simulador TCP.*

3.3 ESTRELLAS FUENTE

Estrictamente hablando, las estrellas fuente son aquellas sin entradas. Sólo generan señales, y pueden representar entradas externas al sistema, datos constantes o estimulación de algún tipo. Por convención, estas estrellas una vez ejecutadas se inician en tiempo cero automáticamente. Durante esta y las demás ejecuciones subsecuentes, la estrella debe determinar por si sola cuando debe ocurrir su siguiente ejecución. En la figura 3.2 se puede apreciar parte de la paleta para las estrellas fuente que intervienen dentro del simulador TCP.



(a)



(b)

Figura 3.2. *Estrellas fuentes en el dominio DE usadas en el simulador TCP. (a)Estrictamente como fuentes y (b) Generan señales.*

- ***Clock***: Genera los eventos a intervalos regulares, comenzando en tiempo cero.

- ***Null***: No hace nada. Este es muy usado para conectarlo a puertos de entrada no utililizables.

- ***Poisson:*** Genera los eventos de acuerdo a un proceso de Poisson. El primero de los eventos se genera en tiempo cero. El tiempo medio de inter-arribo y la magnitud de los eventos se proporcionan como parámetros.

- ***PulseGen:*** Genera los eventos con valores específicos en momentos específicos. Los eventos se especifican en el "arreglo de valores", el cual consiste de pares de tiempo-valor, proporcionados bajo sintaxis de números complejos.

- ***TkButtons:*** ***(dos iconos)*** Saca el valor especificado cuando los botones se presionan. Si el parámetro "*allow_simultaneous_events*" está en YES, entonces los eventos de salida se producen solo cuando el botón marcado con "PUSH TO PRODUCE EVENTS" es presionado. La marca de tiempo de cada evento de salida es puesto al tiempo actual del marcador cuando el botón es presionado.

- ***TkSlider:*** Saca un valor determinado por una escala deslizante interactiva en pantalla.

- ***Const***: Produce un evento a la salida con un valor constante (el valor por default es cero) cuando se simula por un evento de entrada. La marca de tiempo de la salida es exactamente igual a la de la entrada.

- ***Ramp***: Produce un evento de salida con un valor incremental cuando se le simula con otro evento a la entrada. El valor de ese evento de salida comienza en *value* y se incrementa por *step* cada vez que la esta estrella se activa.

- ***RanGen:*** (cuatro iconos) Genera una secuencia de números aleatorios. Al recibir un evento de entrada, brinda un número aleatorio con salida uniforme, exponencial o distribución normal, tal como se haya determinado en el parámetro *distribution.* Dependiendo de la distribución, hay otros parámetros que especifican ya sea la media o varianza, o bien el rango de exceso superior o inferior.

- ***SinGen***: Genera una muestra de una onda seno cuando se dispara. Esta galaxia contiene a una galaxia *singen* del dominio SDF, (es decir es un "hoyo de gusanos")[10].

- ***WaveForm:*** Cuando recibe un evento a la entrada, entonces proporciona el siguiente valor especificado por el parámetro del arreglo *value* (default "1 – 1"). Este arreglo puede ser repetido periódicamente con cualquier período, por lo que se puede detener cualquier simulación cuando se haya alcanzado el final de dicho arreglo. La tabla 3.1 resume las capacidades:

HaltAtEnd	***Periodic***	***Period***	***Operation***
NO	YES	0	El período es igual a la longitud del arreglo.
NO	YES	N>0	El período es N
NO	NO	Anything	Saca el arreglo una vez, luego ceros
YES	Anything	Anything	Se detiene luego de sacar el arreglo una vez.

Tabla 3.1 *Resumen da las capacidades de la estrella WaveForm.*

La primer línea de la tabla, brinda los valores por default. El arreglo debe ser leído desde un archivo simplemente poniendo *value* a algo que tenga la forma <nombre_de_archivo>.

[10] Hoyo de gusanos o "wormhole" es una estrella en algún dominio particular que internamente contiene una galaxia en algún otro dominio.

3.4 ESTELLAS DE SUMIDERO (sink).

Algunas de las estrellas de sumidero apuntadas en la paleta dentro de la figura 3.3 usadas en el simulador TCP son todas aquellas sin salidas. Ellas sólo despliegan señales en varias maneras, o bien las escriben a archivos. Muchas de las estrellas que conforman la paleta completa están basadas en el programa pxgraph. Este programa tiene muchas opciones y las diferencias entre las estrellas suelen sobrepasar un poquito mas que las opciones por default. Sin embargo algunas procesan las señales en formas utilizables antes de pasarlas al programa pxgraph.

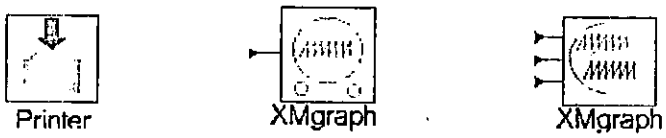


Figura 3.3. *Estrellas de sumidero (sink) en el dominio DE empleadas en el simulador TCP.*

- ***Printer***: Imprime el valor de cada evento que llega, junto con su tiempo de arribo. El parámetro *nombre de archivo* especifica el archivo a ser escrito; los nombres especiales <stdout> y <cout> (especificando la cadena de salida estándar), y <stderr> y <cerr> (especificando la cadena del error estándar), también están contempladas.

- ***Xmgraph (dos iconos)***: Generan una grafica con el programa pxgraph con un punto por evento. Cualquier número de secuencias de evento pueden ser graficados simultáneamente, hasta el límite determinado por la clase de Xgraph. Por default, se dibuja una línea recta entre cada par de eventos.

3.5 ESTRELLAS DE CONTROL.

Las estrellas de control que se muestran en la figura 3.4, manipulan el control de fichas. Todas estas estrellas son polimorfitas; pues operan en cualquier tipo de datos.

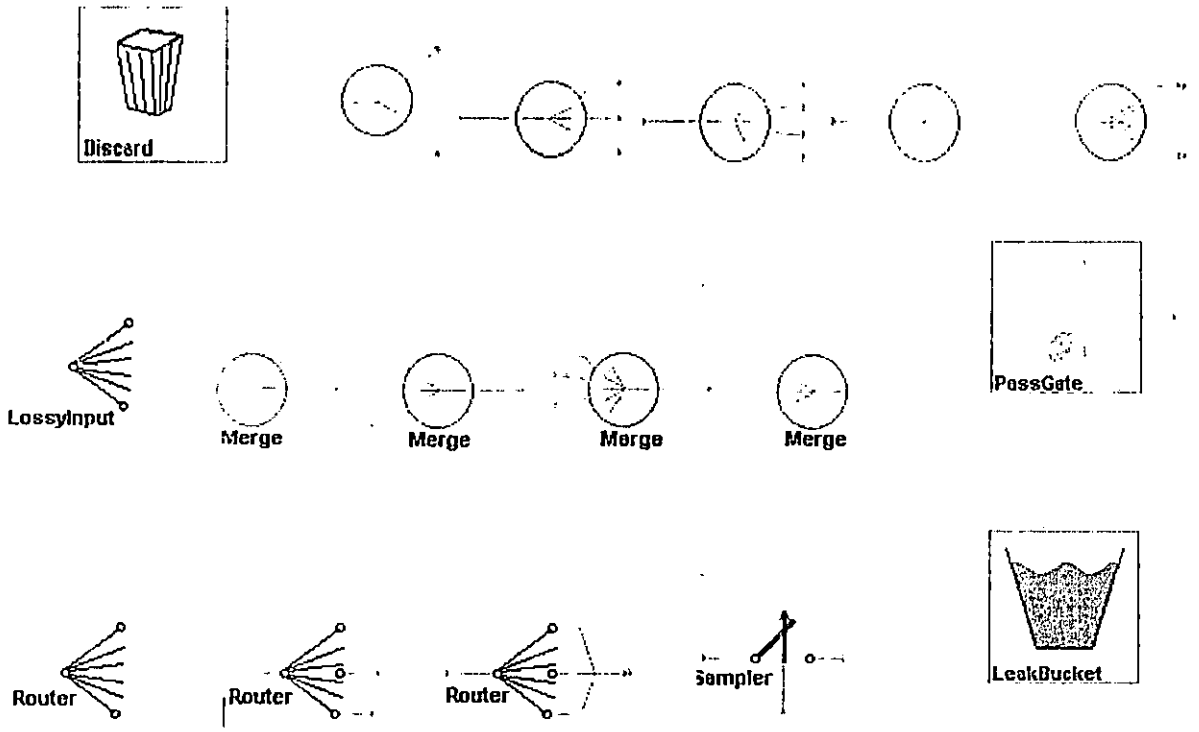


Figura 3.4. *Estrellas de control para el dominio DE dentro de Ptolemy.*

- ***Discard***: Descarta los eventos de entrada que ocurren antes del tiempo de umbral. Los eventos posteriores al tiempo de umbral pasan inmediatamente a la salida. Esta estrella es muy usada para quitar transitorios y estudiar los efectos de estado estable.

- ***Fork(Cuatro iconos)***: Replica los eventos de la entrada a las salidas con cero retardos.

- ***LossyInput***: Rutéa las entradas a la salida de sumidero con la probabilidad *loss probability* puesta por el usuario. Todas las demás entradas son enviadas inmediatamente a la salida *save*.

- ***Merge (Cuatro iconos)***: Fusiona los eventos de entrada, manteniendo orden temporal. Aquellos eventos simultáneos son fusionados en el orden del número de puerto con el cual aparecen, con el puerto # 1 siendo procesado primero.

- ***PassGate***: Si la puerta (entrada inferior) está abierta, entonces las partículas pasan de la "entrada" (entrada izquierda) a la "salida". Cuando la puerta está cerrada, no se produce ninguna salida. Si llegan partículas de entrada mientras la puerta esta cerrada, la mas reciente será pasada a la "salida" cuando la puerta sea reabierta.

- ***Router (Tres iconos)***: rutan un evento de entrada aleatoriamente hacia alguna de sus salidas. La probabilidad es igual para cada salida. El retardo de tiempo es cero.

- ***Sampler***: Muestrea la entrada a tiempos dados por los eventos en la entrada de "reloj". El valor de los datos de la entrada de "reloj" son ignorados. Si no hay ninguna entrada al momento de muestreo, entonces se usa la última entrada. Si no ha habido ninguna entrada, entonces se produce una partícula "cero". El significado exacto de cero depende del tipo de partícula.

- ***LeakBucket***: Descarta las entradas que llegan muy frecuentemente. Es decir, descarta algún evento que pueda causar una cola de un valor dado seguido por un servidor con un rango de servicio establecido a un desbordamiento. Las entradas que no se descartan se pasan inmediatamente a la salida.

3.6 COLAS, SERVIDORES Y RETARDOS.

La paleta que aparece dentro de la figura 3.5, contiene estrellas que modelan colas, servidores y retardos de tiempo de varios tipos. En el dominio DE, el icono de retardo (el que aparece en la esquina superior izquierda) no representa un retardo de tiempo.

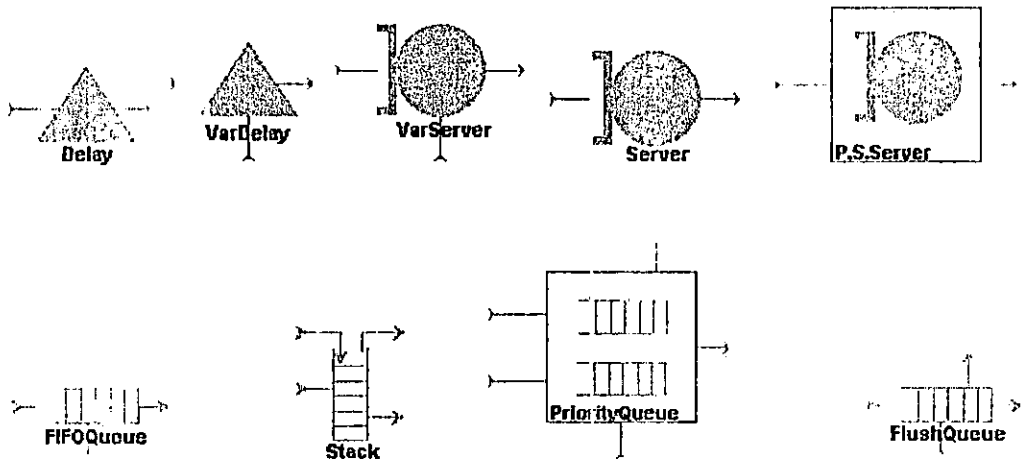


Figura 3.5. *Colas, Servidores y Retardos en el dominio DE dentro de Ptolemy.*

- ***Delay***: Envía cada evento de entrada hacia la salida con su marca de tiempo incrementado por un monto dado por el parámetro de *retardo*.

- ***VarDelay***: Retarda la entrada por un monto variable. El parámetro de *retardo* proporciona el retardo inicial, y el retardo se cambia usando la entrada "nuevo retardo".

- ***PSServer***: Emula ser un determinante servidor de procesador-compartido. Si los eventos de entrada llegan cuando no esta ocupado, entonces los retarda con el tiempo de servicio nominal. Si llegan cuando está ocupado, entonces el servidor es compartido. Por tanto los arribos primarios que aún están en servicio serán retrasados por mas del tiempo de servicio nominal.

- ***Server***: Emula a un servidor. Si los eventos de entrada llegan cuando no está ocupado, entonces los retrasa por el tiempo de servicio (parámetro constante). Si llegan cuando está ocupado, entonces los retarda por el tiempo de servicio mas el tiempo que tome liberarse de tareas previas.

- ***VarServer***: Emula un servidor con un tiempo de servicio variable. Si los eventos de entrada llegan cuando no está haciendo nada, entonces serán atendidos inmediatamente y sólo será retrasado por el tiempo de servicio. Si los eventos de entrada llegan mientras otro evento está siendo atendido, serán puestos en cola. Cuando el servidor se libera, atenderá cualquier evento esperando en su cola.

- ***FIFOQueue***: Implementa una cola del tipo primero en entrar, primero en salir (FIFO) de longitud finita o infinita. Los eventos que están en la entrada de "demanda" disparan otra cola en el puerto "outData" si la cola no está vacía. Si esta está vacía, entonces un evento en "demanda" habilita la próxima partícula "inData" para que pase inmediatamente hacia "outData". La primer partícula que llega al "inData" siempre es conducido directamente a la salida, a menos que el *numDemandsPending* esté inicializado a 0. Si *consolidateDemands* esta puesto en TRUE (default), entonces *numDemandsPending* no se le permite que suba arriba de uno. El tamaño de la cola es mandado a la salida "size" siempre que un evento "inData" o en "demand" es procesado. Los datos de entrada que no caben en la cola son enviados a la salida "overflow".

- ***FlushQueue***: Implementa una cola FIFO la cual cuando se encuentra llena, descarta todas las entradas hasta que se vacía completamente.

- ***PriorityQueue***: Emula una cola prioritaria. Las entradas tienen prioridad de acuerdo al número de entradas, con mayor prioridad "inData#1" , después "inData#2", etc. Cuando se recibe un "demand", las salidas se producen seleccionando la selección basada en la prioridad, y luego basado en tiempo de llegada, usando un sistema

FIFO. Una capacidad total finita puede especificarse al poner el parámetro de capacidad a un valor entero. Cuando se alcanza la capacidad, las entradas posteriores son enviadas a una salida "overflow", y no son almacenadas. Los parámetros *numDemandPending* y *consolidateDemands* tienen el mismo significado como en otras estrellas de colas. El tamaño de la cola es enviado a la salida "size" en cualquier momento que sea procesado un evento "inData" o en "demand".

- ***Stack***: Implementa una pila ya sea de longitud finita o infinita. Los eventos a la entrada "demend" mandan datos de la pila hacia "outData" si la pila no esta vacía. Si esta vacía, entonces un evento en "demand" habilita la próxima partícula para que pase inmediatamente hacia "outData". Por omisión, *numDemandPending* se inicializa con 1, asi que la primer partícula a llegar "inData", es pasado directamente a la salida. Si *consolidateDemands* esta puesta en TRUE (default), entonces *numDemandsPending* no se le permite que suba a uno. El tamaño de la pila es enviado al tamaño de la salida "size" cuando un evento "inData" o "demand" es procesado. Los datos de entrada que no caben en la pila son mandados a la salida "overflow".

3.7 CONCLUSIÓN

El corazón de Ptolemy es una estructura de software compacto sobre el cual existen entornos de diseño especializados (llamados *dominios*) que pueden ser construidos. La infraestructura de software, llamada *el Corazón de Ptolemy*, esta constituido por definiciones de clases en C++. Los dominios se definen creando nuevas clases de C++ derivadas de las clases básicas dentro del corazón.

Los dominios pueden operar en cualquiera de las dos formas:

- Simulación: Un organizador invoca segmentos de código en un orden apropiado al modelo de computación.

- Generación de Código: Segmentos de código en un lenguaje arbitrario se juntan para producir uno o mas programas que implementen la función especificada.

El uso de una tecnología de software orientada a objetos le permite a un dominio a interactuar uno con el otro sin conocimiento de las características o semánticas del otro. Por eso, al emplear una variedad de dominios, un equipo de diseñadores puede modelar cada subsistema de manera eficiente. Todos estos subsistemas diferentes pueden ser anidados para constituir un árbol de subsistemas. Esta composición jerárquica es clave al especificar, simular y sintetizar sistemas complejos hetero geniales.

En pocas palabras, Ptolemy es una fundación flexible sobre la cual es posible construir entornos de prototipos, que cuenta, por ejemplo, con programación gráfica orientada al flujo de datos, para procesamiento de datos, modelado de redes con procesos para multi-tarea, modelado de eventos discretos para redes de comunicación, etc.

CAPITULO 4

"El Simulador TCP"

4.1 INTRODUCCIÓN.

El simulador consiste de dos componentes que simplemente pueden ser conectados a los sistemas terminales de la red. Las versiones del TCP tanto básica y en las versiones de retransmisión rápida posteriores tienen estructuras idénticas y por ello pueden ser intercambiables sin tener que modificar el entorno en el que estén siendo utilizadas.

4.2 EL TRANSMISOR TCP.

Tal como se puede apreciar de la figura 4.1, el transmisor consta de tres partes principales que utiliza el protocolo TCP para conformar el lado de transmisión. Este circuito está constituido por una galaxia la cual contiene dos entradas y una salida. Una de las terminales de entrada es aquella por la cual recibe la información sobre el tamaño de los paquetes a ser enviados y la otra es por donde recibe los acuses de recibo correspondientes a cada paquete que se envía. Por el otro lado la terminal que tiene de salida es precisamente donde se transmiten los paquetes una vez que se han completado dentro de esta galaxia.

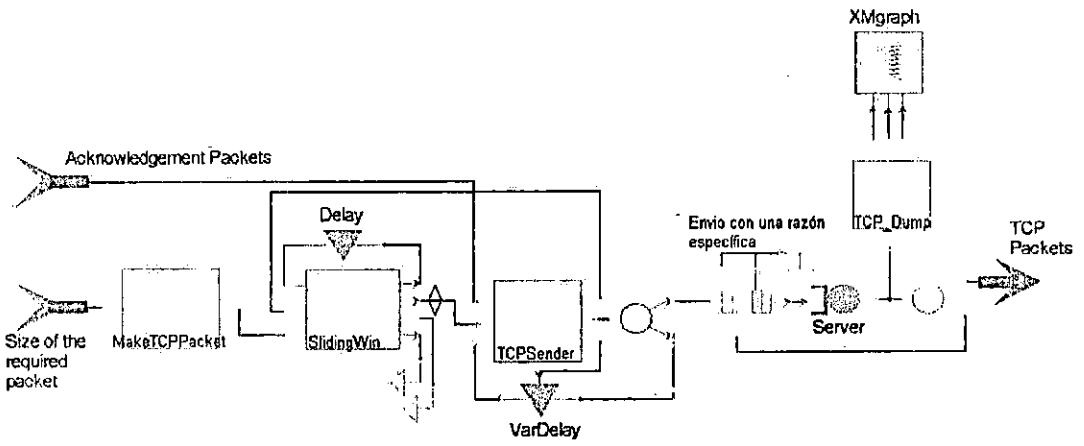


Figura 4.1. *El Modelo del transmisor TCP.*

A continuación se brinda una breve descripción de las principales estrellas contenidas en la galaxia que conforma la etapa de transmisión así como su respectiva función.

4.2.1 MakeTCPPacket:

Cuando el transmisor del sistema desea enviar un paquete TCP, se le debe informar a ésta estrella sobre el tamaño de dicho paquete a ser enviado. Después de todo esto, se genera el paquete conteniendo los campos de información necesaria como lo son las direcciones tanto del destino y la fuente, el número de secuencia de octeto, el tamaño de los paquetes y el número de secuencia de paquete. Dado que estos campos están siendo completados antes de conducir al paquete hacia la siguiente estrella, el campo designado para los acuses de recibo permanece vacío, ya que será utilizado por el receptor cuando el paquete llegue hasta el. En la figura 4.2 se puede apreciar la identidad de sus terminales.

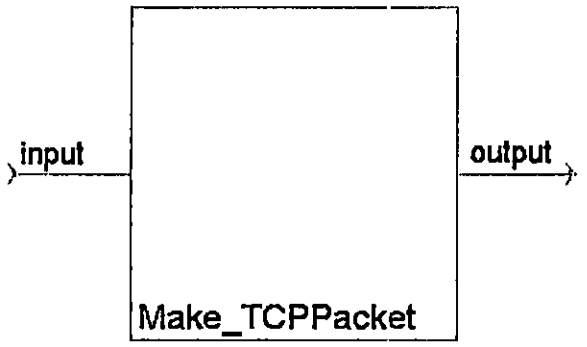


Figura 4.2. *Estrella MakeTCPPacket con los nombres para sus terminales.*

La *MakeTCPPacket* es una estrella ligada dinámicamente con la galaxia del transmisor que fue creada dentro del dominio de los eventos discretos (DE) de Ptolemy. Tal como se mencionó antes, esta estrella genera un paquete TCP vacío al ejecutarla. El paquete empleado para la simulación es una estructura de datos consistente en la información para los campos de la fuente, destino, acuses de recibo, tamaño, número de secuencia de octeto y número de secuencia. Como entrada recibe el tamaño del paquete.

Los detalles sobre la clase de datos que emplea la *MakeTCPPacket* así como la configuración de sus variables se presentan a continuación en la tabla 4.1:

Entradas	
Entrada 1 (*input*)	*Valores de tipo: FLOTANTE*
Salidas	
Salida 1 (*output*)	*De tipo MENSAJE*
Estados Configurables	
Start_Value	*Valores del tipo entero, que constituye el valor de inicio para el contador de secuencia. Por default la estrella MakeTCPPacket pone un valor de 0.*
Source	*Valores del tipo entero, e indica el número del nodo de la fuente. Por default la*

	estrella MakeTCPPacket utiliza un valor igual a 0.
Destination	*Valores del tipo entero, e indica el número del nodo del destino. Por default la estrella MakeTCPPacket utiliza un valor igual a 0.*
VC	*Valores del tipo entero, e indica el número del nodo de destino. Por default la estrella MakeTCPPacket utiliza un valor igual a 0.*

Tabla 4.1. *Clase de datos y valores configurables para la estrella MakeTCPPacket.*

4.2.2 **SlidingWindow:**

Esta estrella no es otra cosa que una cola FIFO (First in-First out) que funciona como la ventana deslizante en el protocolo TCP. Tal como se aprecia en la figura 4.1 posee dos terminales de entrada: una que es por donde se reciben los paquetes provenientes de la estrella MakeTCPPacket y la otra por la cual el sistema transmisor actual marca el número del último octeto que puede ser enviado antes de que se cierre la ventana.

También existe una tercer terminal de entrada para cuando llega alguna señal que permite a la estrella *SlidingWin* dar inicio al envío de los paquetes, ya que puede darse la posibilidad de que se manden mas de un paquete simultáneamente. Por lo tanto, se ha introducido un retardo entre los paquetes subsecuentes. Esto evita el envío de paquetes con la misma marca de tiempo. Con este artificio, se logra conseguir que represente ya sea la tasa de envío o bien el tiempo mínimo de retardo entre paquete y paquete. En la figura 4.3 se presentan las terminales con sus respectivos nombres.

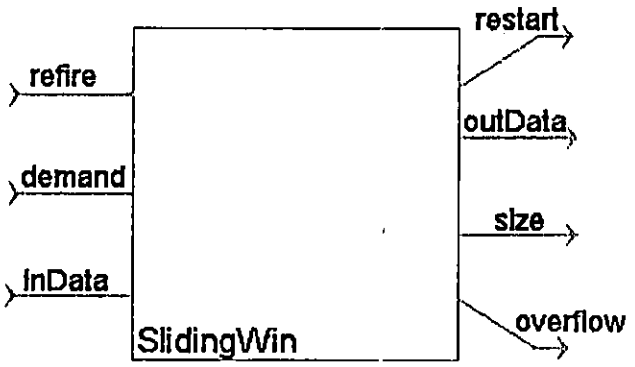


Figura 4.3. *Estrella SlidingWin con los nombres para sus terminales.*

La *SlidingWindow* es también una estrella dinámicamente vinculada con la galaxia que conforma al transmisor dentro del dominio de eventos discretos de Ptolemy (DE) que se encarga de recibir los paquetes que entran y los envía hacia la fuente (la *TCPSender* que se encarga de mandarlos por el puerto de salida del transmisor) si acaso es posible, es decir, si la ventana de transmisión aún no está llena. De otra manera, almacena los paquetes y espera hasta que la fuente envíe una notificación, en la cual se informa que la ventana se ha movido y que por lo tanto es posible enviar datos. En este caso, sólo manda el número de octetos permitidos.

Los detalles sobre la clase de datos que emplea la *SlidingWindow* así como la configuración de sus variables se presentan a continuación en la tabla 4.2:

Entradas	
Entrada 1 (*refire*)	*Valores de tipo: FLOTANTE*
Entrada 2 (*demand*)	*Valores de tipo: ENTERO*
Entrada 3 (*inData*)	*Valores de tipo: CUALQUIERA*
Salidas	
Salida 1 (*restart*)	*De tipo MENSAJE*
Salida 2 (*outData*)	*Valores de tipo: CUALQUIERA*
Salida 3 (*size*)	*Valores de tipo: ENTERO*
Salida 4 (*overflow*)	*Valores de tipo: CUALQUIERA*

Estados Configurables	
Capacity	*Valores del tipo entero, y establece el tamaño máximo de la cola. Si éste valor es menor que cero, entonces la capacidad es infinita. Por default, el valor que se emplea es de 1.*
MSS	*Valores del tipo entero que definen el tamaño máximo de segmento. Por default se utiliza un valor de 1*
Rate	*Valores del tipo flotante que definen la razón de envío del transmisor. Por default se emplea un valor de 1.*
FileName	*Valores del tipo string con el cual se define el nombre del archivo para la salida. Por default se usa la salida estándar <stdout>*

Tabla 4.2. *Clase de datos y valores configurables para la estrella SlidingWindow*

4.2.3 TCPSender:

Esta estrella constituye el mecanismo de envío actual propiamente dicho del transmisor y en algunas ocasiones se le hace referencia como *la fuente*. Los acuses de recibo que llegan y la ejecución de los diferentes servicios provistos por el protocolo TCP habilitan la acción de esta estrella para que determine el numero de octetos permitidos que pueden ser enviados y si hay necesidad de retransmisión.

Puede haber alguna diferencia en versiones diferentes de simuladores, aunque básicamente el único cambio que puede haber en cuanto a la implementación es que en la versión mas básica, esta estrella tiene que retransmitir primero todos los paquetes que se almacenaron en la ventana local luego de una retransmisión. Sólo

hasta entonces es que se pueden enviar nuevos datos. En tanto que para una versión mas extensa, por otro lado, los datos nuevos pueden tomarse fuera de la ventana deslizante y ser enviados directamente luego de recibir el respectivo acuse de recibo para el paquete que se haya perdido y todos aquellos que hayan sido enviados después de el. En la figura 4.4 se muestran sus terminales.

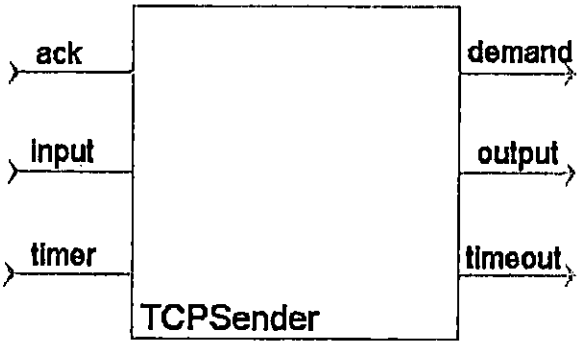


Figura 4.4. *Estrella TCPSender con los nombres para sus terminales.*

La *TCPSender* es también una estrella que está dinámicamente vinculada dentro del transmisor y que opera bajo el dominio de eventos discretos (DE) de Ptolemy. Esta es a la cual nos hemos referido antes como la fuente y que está basada en la versión mas básica de implementación del protocolo TCP y cumple con los siguientes requerimientos:

- *Mecanismo de Ventana Deslizante:* Sólo envía si el receptor cuenta con suficiente espacio. De otra manera, espera hasta que el receptor haya liberado alguna localidad y haya reabierto la ventana de retransmisión.
- *Expiración de Tiempo:* Si un acuse de recibo para algún paquete específico no ha llegado luego de algún tiempo, se le retransmite.
- *Estimación de Viaje Redondo:* Esto se lleva a cabo mediante el empleo del algoritmo de Karn

- *Arranque Lento:* Utilizado para el comienzo de la transmisión y después de alguna expiración de tiempo.
- *Evasión de Congestionamiento:* No es otra cosa que un incremento lento del tamaño de la ventana de 1/cwnd.
- *Retransmisión Rápida:* después de recibirse tres ack's duplicados, la fuente retransmite el paquete perdido.

Los detalles sobre la clase de datos que emplea la *TCPSender* así como la configuración de sus variables se presentan a continuación en la tabla 4.3:

Entradas	
Entrada 1 (*ack*)	*Valores de tipo: MENSAJE*
Entrada 2 (*input*)	*Valores de tipo: MENSAJE*
Entrada 3 (*timer*)	*Valores de tipo: MANSAJE*
Salidas	
Salida 1 (*demand*)	*Valores de tipo ENTERO*
Salida 2 (*output*)	*Valores de tipo: MENSAJE*
Salida 3 (*timeout*)	*Valores de tipo: ENTERO*
Estados Configurables	
Win_Size	*Valores del tipo entero, y establece la ventana de transmisión máxima. Por default, el valor que se emplea es de 10.*
MSS	*Valores del tipo entero que definen el tamaño máximo de segmento. Por default se utiliza un valor de 1*
init_time	*Valores del tipo flotante y que representan el valor inicial para RTT. Por default se toma un valor de 1.*
	Valores del tipo flotante que definen la

Rate	*razón de envío de la fuente. Por default se emplea un valor de 1.*
Grain	*Valores del tipo flotante con el cual se define el número de puntos de reloj en un segundo. Por default se establece un valor de 2.*
Debug	*Valores del tipo entero, conel que se activa o desactiva el iforme.*
FileName	*Valores del tipo string con el cual se define el nombre del archivo para la salida. Por default se usa la salida estándar <stdout>*

Tabla 4.3. *Clase de datos y valores configurables para la estrella TCPSender.*

4.3 EL RECEPTOR TCP

La parte receptora del simulador se responsabiliza de generar los acuses de recibo o reconocimientos (ack's), la detección en la pérdida de paquetes y de simular al usuario. Todo ello se lleva a cabo por medio de las principales estrellas que aparecen en la figura 4.5 y que se describen mas adelante.

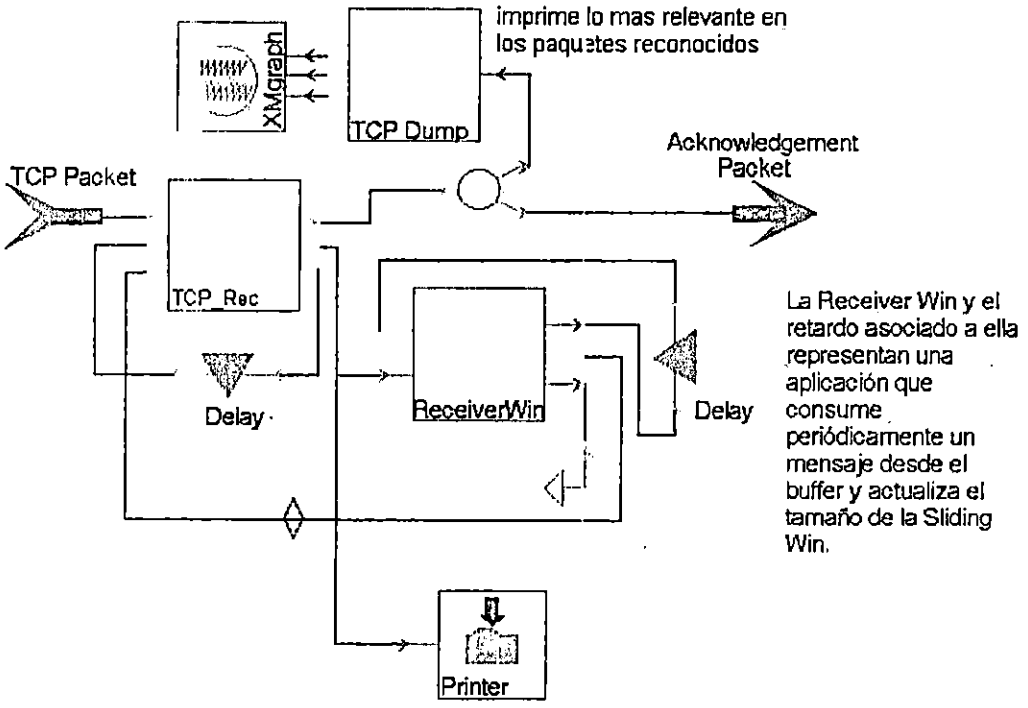


Figura 4.5. *Modelo del Receptor TCP.*

A continuación se brinda una breve descripción de las principales estrellas contenidas en la galaxia que conforma la etapa de recepción así como su respectiva función.

4.3.1 **TCP_Rec:**

Esta estrella constituye el receptor propiamente dicho, y es quien se encarga de generar los acuses de recibo cuando han llegado los paquetes de datos y también se encarga de duplicar los acuses de recibo cada vez que se reciban paquetes que estén fuera de orden. En el caso de haber retransmisión rápida, esta estrella debe además almacenar aquellos paquetes que estén fuera de orden hasta que el paquete perdido sea retransmitido. Sólo hasta entonces podrán pasar todos esos paquetes hasta el usuario. En la figura 4.6 se identifican las terminales para la estrella *TCP_Rec.*

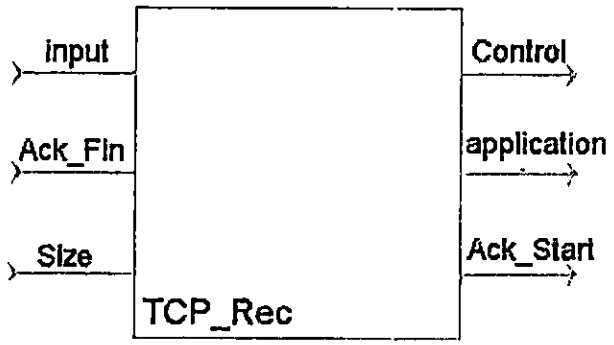


Figura 4.6. *Estrella TCP_Rec con los nombres para sus terminales.*

La TCP_Rec es también una estrella enlazada dinámicamente dentro del dominio de eventos discretos de Ptolemy que simula a un receptor TCP y que además cumple con los siguientes requerimientos:

- Ventana Deslizante: Al emplear la técnica de acuses de recibo para los paquetes, el receptor informa a la fuente sobre el número de octetos permitidos que pueden ser enviados antes de que el buffer del transmisor logre llenarse.
- Actualización de Paquetes: Después de haber enviado un acuse de recibo con el tamaño de ventana menor al tamaño máximo de segmento (MSS), se envía un paquete de actualización cuando haya espacio nuevo disponible en el buffer.
- Ventana Tonta: Si el tamaño de la ventana se hace muy pequeño, (mas pequeño que el tamaño máximo de segmento MSS), entonces se envía un paquete de actualización únicamente hasta después de haber liberado suficiente espacio en el buffer. En este caso, dentro del simulador se ha configurado para que sea la mitad del tamaño de ventana original.
- Duplicado de los Acuses de Recibo: Si el paquete se pierde en el camino, el receptor envía para cada paquete nuevo recibido un acuse de recibo del último paquete correcto que haya llegado. Todos los paquetes fuera de orden son colocados en un buffer hasta que el paquete que hace falta en la secuencia se reciba. Sólo hasta entonces podrán ser reconocidos todos estos otros paquetes.

- Paquetes Retardados: Con esta técnica, los paquetes reconocidos se envían luego de que expira un temporizador de reconocimientos o hasta que cualquier otro paquete llegue.

En este modelo, se asume que todos los paquetes poseen la misma longitud que corresponde a la longitud máxima de segmento MSS. Basado en todo esto, el receptor solamente informa a la aplicación cuando un paquete es recibido. Si existe algún interés en emplear tamaños de paquete variables, entonces el paquete actual recibido deberá enviarse a la aplicación.

Los detalles sobre la clase de datos que emplea la *TCP_Rec* así como la configuración de sus variables se presentan a continuación en la tabla 4.4:

Entradas	
Entrada 1 (input)	*Valores de tipo: MENSAJE*
Entrada 2 (Ack_Fin)	*Valores de tipo: ENTERO*
Entrada 3 (Size)	*Valores de tipo: ENTERO*
Salidas	
Salida 1 (Control)	*Valores de tipo MENSAJE*
Salida 2 (application)	*Valores de tipo: ENTERO*
Salida 3 (Ack_Start)	*Valores de tipo: ENTERO*
Estados Configurables	
Window_Size	*Valores del tipo entero, y establece la ventana de transmisión máxima. Por default, el valor que se emplea es de 10.*
FileName	*Valores del tipo string con el cual se define el nombre del archivo para la salida. Por default se usa la salida estándar <stdout>*

MSS	*Valores del tipo entero que definen el tamaño máximo de segmento. Por default se utiliza un valor de 1*
Ack_time	*Valores del tipo entero y que representan el valor de inicio para el temporizador de acuses de recibo. Por default se toma un valor de 1.*
Debug	*Default = 0.*

Tabla 4.4. *Clase de datos y valores configurables para la estrella TCP_Rec.*

4.3.2 **ReceiverWin:**

Esta estrella representa a un usuario de TCP que consume un paquete a intervalos regulares de tiempo. En el presente simulador, ha sido construido como una cola FIFO con auto disparo. Cuando llega el primer paquete, este es transmitido directamente hacia una estrella de retardo. Todos los siguientes paquetes son entonces puestos en cola. Cuando el primero de los paquetes se introduce a la estrella, es descartado por lo que un nuevo paquete puede abandonar la cola.

Cuando un paquete es agregado o removido de la cola, se envía un mensaje hacia la *TCP_Rec* que contiene el tamaño actual de la cola. Con esta información el receptor puede determinar el tamaño apropiado de la ventana de advertencia. En la figura 4.7 se aprecian las terminales de la estrella *ReceiverWin.*

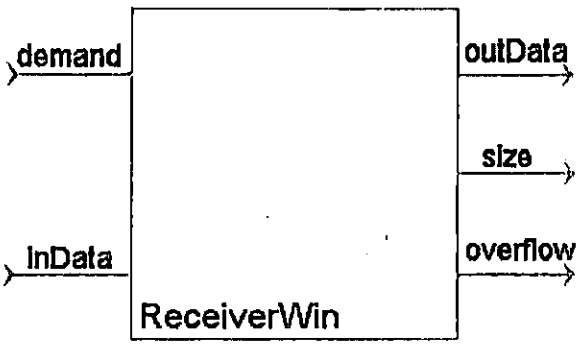


Figura 4.7. *Estrella ReceiverWin con los nombres para sus terminales.*

La estrella *ReceiverWin* es también otra estrella que esta dinámicamente enlazada con el sistema, operando dentro del dominio de eventos discretos de Ptolemy y simula a un buffer para la aplicación. Se asume que los paquetes siempre tienen un tamaño constante e igual al tamaño máximo de segmento MSS. El buffer solamente recibe el número de secuencia proveniente del receptor TCP.

Cuando la aplicación esta lista para un nuevo paquete, el tamaño de la ventana es cambiado por el tamaño máximo de segmento MSS. Si existe algún interés en tamaños variables de paquete, entonces el receptor debe ser modificado para que envíe paquetes reales.

Los detalles sobre la clase de datos que emplea la *ReceiverWin* así como la configuración de sus variables se presentan a continuación en la tabla 4.5:

Entradas	
Entrada 1 (demand)	*Valores de tipo: ENTERO*
Entrada 2 (inData)	*Valores de tipo: CUALQUIERA*
Salidas	
Salida 1 (outData)	*Valores de tipo: CUALQUIERA*
Salida 2 (size)	*Valores de tipo: ENTERO*
Salida 3 (overflow)	*Valores de tipo: CUALQUIERA*
Estados Configurables	
Capacity	*Valores del tipo entero, y establece el tamaño máximo de la cola. Si es menor que cero, la capacidad es infinita. Por default, el valor que se emplea es de -1.*
MSS	*Valores del tipo entero que definen el tamaño máximo de segmento. Por default se utiliza un valor de 1*
Ack_time	*Valores del tipo entero y que representan el valor de inicio para el temporizador de acuses de recibo. Por default se toma un valor de 1.*

Tabla 4.5. *Clase de datos y valores configurables para la estrella ReceiverWin.*

4.3.3 TCP_Dump:

También existe otra estrella que utilizan ambas galaxias tanto del transmisor como del receptor, llamada *TCP_Dump*.

La estrella *TCP_Dump* es también otra estrella ligada dinámicamente al sistema creada dentro del dominio de eventos discretos de Ptolemy y se encarga de leer dentro de un paquete TCP y poner a la salida su contenido.

Los detalles sobre la clase de datos que emplea la *TCP_Dump* así como la configuración de sus variables se presentan a continuación en la tabla 4.6:

Entradas	
Entrada 1 (input)	*Valores de tipo: MENSAJE*
Salidas	
Salida 1 (seq)	*Valores de tipo: ENTERO*
Salida 2 (ack)	*Valores de tipo: ENTERO*
Salida 3 (Win)	*Valores de tipo: ENTERO*
Estados Configurables	
Logfile	*Valores del tipo string. Default =*

Tabla 4.6. *Clase de datos y valores configurables para la estrella TCP_Dump.*

4.4 MODELO DE RED.

La figura 4.8 presenta un modelo topológico de una red que ya antes ha sido empleado en otros estudios similares al presentado aquí y que ha demostrado simular con mucha certeza una red real. Un paso crucial y frecuentemente olvidado que debe hacerse con gran precisión es verificar el modelo simulado y por ello, el modelo de red empleado aquí redujo los requerimientos de verificación a una simple comparación entre los resultados obtenidos actualmente y los ya realizados en otros estudios.

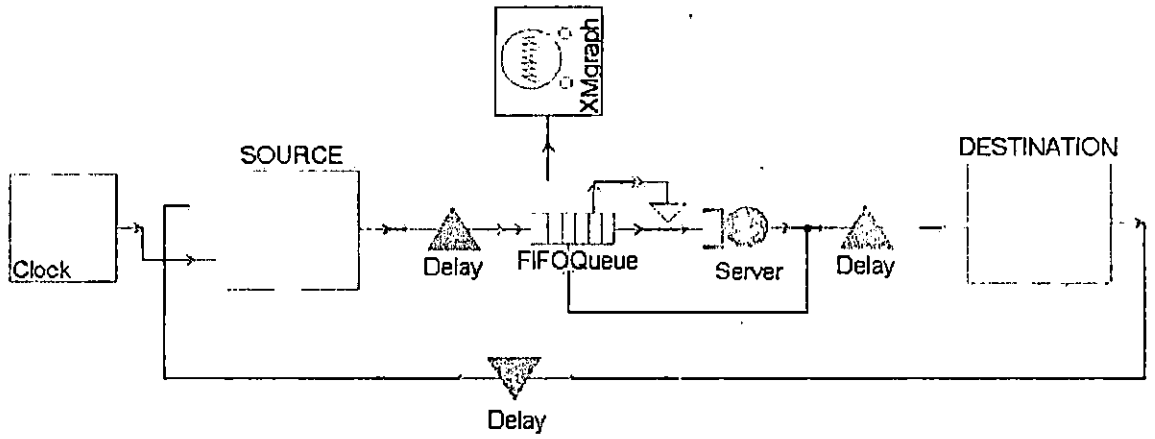


Figura 4.8. *Modelo de la red dentro del simulador en Ptolemy.*

La topología de red elegida es bastante fiel como puede apreciarse de la figura 4.8. una cola FIFO representando un búfer cuello de botella se conecta entre el transmisor y el receptor. El otro cuello de botella de transmisión se ubica entre la cola FIFO y el destino, representado principalmente por un Server y un Delay que proporcionen un retardo de propagación τ (el cual se puede poner en 10 milisegundo). La línea de transmisión entre la fuente y la cola FIFO representada solo por un Delay, presenta un retardo de propagación que puede ser ajustado a 1 milisegundo.

4.5 CONCLUSION

De todo lo expuesto previamente, se puede tratar de hacer un enlace entre los servicios que proporciona el TCP con el modelo de simulación bajo ambiente Ptolemy. Hasta ahora solo se ha hablado de las principales características que presenta el simulador y de cómo es posible controlar sus diferentes parámetros y obtener resultados describiendo el normal flujo de paquetes y retransmisión de los mismos en caso de pérdida bajo condiciones de tráfico o por expiración en el tiempo de viaje redondo, el siguiente paso será el análisis de los resultados de simulación que sin duda alguna representan el control del transporte con el TCP.

CAPITULO 5

"Resultados de Simulación"

5.1 INTRODUCCION

Antes de obtener los resultados del sistema, es necesario haber definido de manera ordenada y concisa todos los parámetros necesarios que se sugieren en al capitulo anterior. El objetivo de esto es probar las características descritas a lo largo del estudio del TCP, verificar los servicios que presta el protocolo de transporte de datos de manera fiel con lo esperado. Las posibilidades que deben haber pueden ser tanto la retransmisión por perdida de paquetes, por expiración en el tiempo de viaje redondo (RTT), la secuencia de los paquetes, etc.

5.2 RESULTADOS BAJO CONDICIONES NORMALES.

La manera de apreciar una simulación en la que obtengamos resultados bajo condiciones de transporte normales, puede realizarse al proporcionar los siguientes parámetros de simulación:

PARA EL TRANSMISOR

MakeTCPPacket		***SlidingWin***	
Start_value	= 0	capacity	= -1
source	= 0	MSS	= 1
destination	= 0	Rate	= 0.01
VC	= 0	filename	= /root/uno.ascii
		Delay	= 0.01

TCPSender

Win_Size = 10
MSS = 1
Init_time = 1
Rate = 0.01
Grain = 2
Debug = 1
Filename = /root/dos.ascii
Delay = 0.1

Otras Estrellas

FIFOQUEUE:
Capacity = 3
numDemandPending = 1
ConsolidateDemands = TRUE
SERVER:
ServiceTime = 0.1
%DELAY:
Delay = 0.1

PARA EL RECEPTOR

TCP_Rec

Window_Size = 5
Filename = /root/tres.ascii
MSS = 1
Ack_Time = 1
Debug = 1

ReceiberWin

Capacity = -1
MSS = 1
Delay = 0.01

Otra Estrellas

DELAY:
Delay = 0.1
%DELAY:
delay = 1
PRINTER:
Filename = /root/cuatro.ascii

PARA EL CANAL DE TRANSMISIÓN (red)

Otras Estrellas

CLOCK:
Interval = 0.01
Magnitude = 1.0
SERVER:
Service_Time = 1
FIFOQUEUE:
Capacity = -1
NumDemandsPending = 1
ConsolidateDemands = TRUE

DELAY (#1):
delay = 0.01

DELAY (#2):
delay = 0.01

DELAY (#3):
delay = 1.0

Utilizando un proceso de ejecución en Ptolemy para 50 puntos de iteración, los resultados gráficos obtenidos en las estrellas de sumidero (XMgraph) que están ubicadas a lo largo del diagrama en el circuito que se presentó en las figuras 4.1, 4.5 y 4.8 para el transmisor, receptor y la red respectivamente, se ilustran a continuación en las figuras 5.1, 5.2 y 5.3 respectivamente:

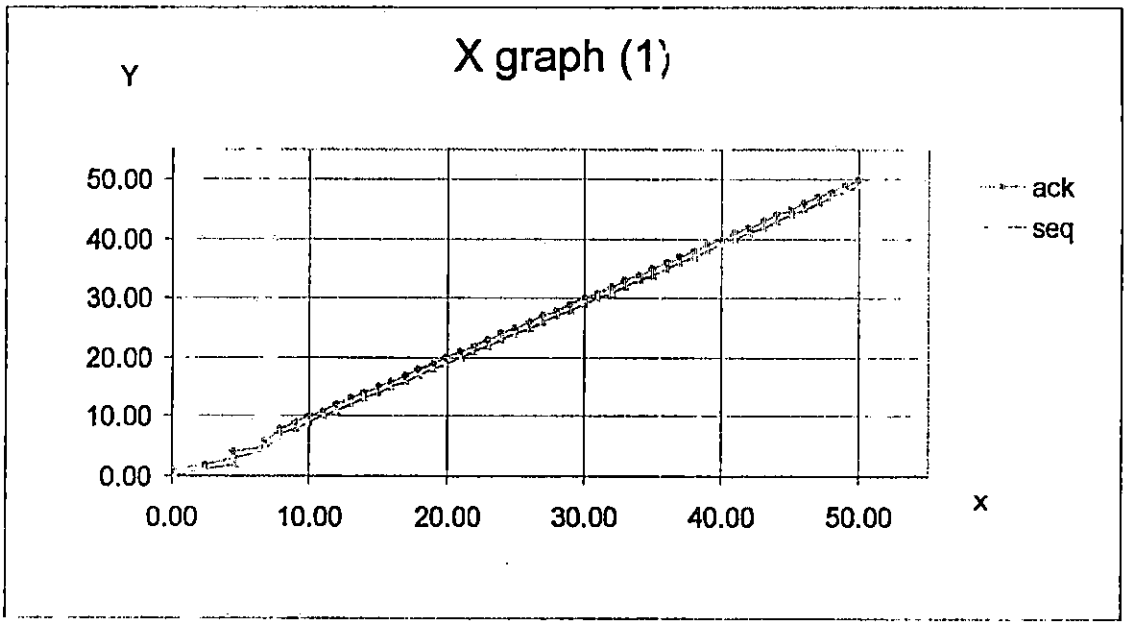


Figura 5.1. *Resultados gráficos a la salida del transmisor TCP y que representan al flujo de paquetes enviados hacia la red con destino al receptor, correspondientes a XMgraph (1) según el diagrama del circuito.*

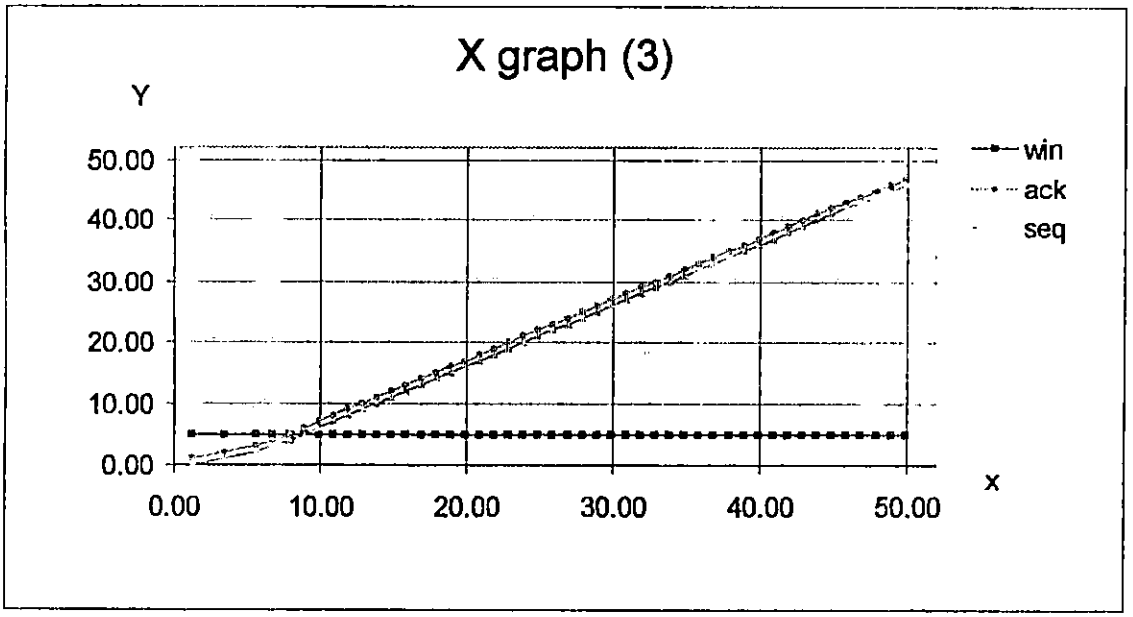


Figura 5.2. *Resultados gráficos a la salida del receptor TCP y que representan el flujo de los acuses de recibo hacia el transmisor, correspondientes a XMgraph (3) según el diagrama del circuito.*

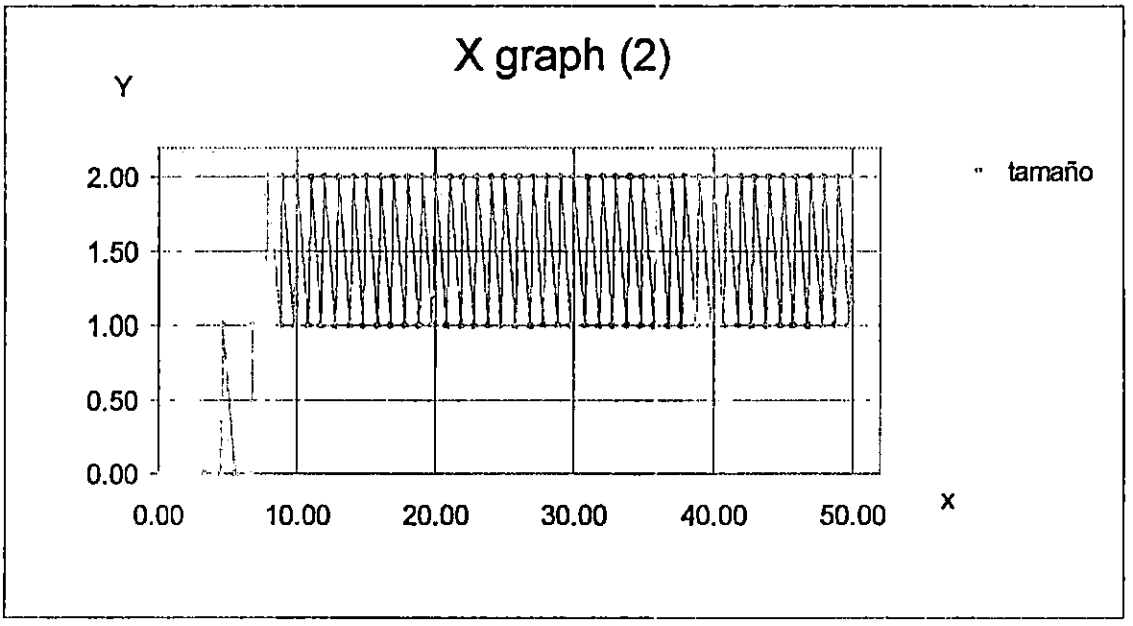


Figura 5.3. *Resultados gráficos que determinan el tamaño de la cola FIFO dentro del modelo de red y que representan el flujo de paquetes a través de ella, correspondientes a XMgraph (2) según el diagrama del circuito.*

En las gráficas anteriores puede observarse claramente el tráfico de los paquetes a través del sistema, proporcionando información continua del tamaño de ventana (win), acuses de recibo (ack), números de secuencia (seq) y en la red el tamaño (size) de la cola. Esta y otra información adicional puede también apreciarse en los archivos generados por las estrellas *SlidingWin, TCPSender, TCP_Rec y Printer* y que fueron almacenados según la ruta asignada en los parámetros que se describieron arriba con los nombres /root/uno.ascii, /root/dos.ascii, /root/tres.ascii y /root/cuatro.ascii respectivamente. Esta información adicional para este ejemplo bajo condiciones normales de transmisión se presenta a continuación en los cuadros 5.1, 5.2, 5.3 y 5.4:

```
NEW RESTART
refire at    2.22
OUT Size  3     Rate  0.01  at  2.22
refire at    4.44
OUT Size  5     Rate  0.01  at  4.44
OUT Size  5     Rate  0.01  at  4.45
refire at    6.66
OUT Size  7     Rate  0.01  at  6.66
```

OUT Size	7	Rate	0.01	at	6.67
refire at	7.66				
OUT Size	9	Rate	0.01	at	7.66
OUT Size	9	Rate	0.01	at	7.67
refire at	8.88				
OUT Size	10	Rate	0.01	at	8.88
refire at	9.88				
OUT Size	11	Rate	0.01	at	9.88
refire at	9.88				
OUT Size	12	Rate	0.01	at	10.88
refire at	10.88				
OUT Size	13	Rate	0.01	at	11.88
refire at	11.88				
OUT Size	14	Rate	0.01	at	12.88
refire at	12.88				
OUT Size	15	Rate	0.01	at	13.88
refire at	13.88				
OUT Size	16	Rate	0.01	at	14.88
refire at	14.88				
OUT Size	17	Rate	0.01	at	15.88
refire at	15.88				
OUT Size	18	Rate	0.01	at	16.88
refire at	16.88				
OUT Size	19	Rate	0.01	at	17.88
refire at	17.88				
OUT Size	20	Rate	0.01	at	18.88
refire at	18.88				
OUT Size	21	Rate	0.01	at	19.88
refire at	19.88				
OUT Size	22	Rate	0.01	at	20.88
refire at	20.88				
OUT Size	23	Rate	0.01	at	21.88
refire at	21.88				
OUT Size	24	Rate	0.01	at	22.88
refire at	22.88				
OUT Size	25	Rate	0.01	at	23.88
refire at	23.88				
OUT Size	26	Rate	0.01	at	24.88
refire at	24.88				
OUT Size	27	Rate	0.01	at	25.88
refire at	25.88				
OUT Size	28	Rate	0.01	at	26.88
refire at	26.88				
OUT Size	29	Rate	0.01	at	27.88
refire at	27.88				
OUT Size	30	Rate	0.01	at	28.88

refire at	28.88				
OUT Size	31	Rate	0.01	at	29.88
refire at	29.88				
OUT Size	32	Rate	0.01	at	30.88
refire at	30.88				
OUT Size	33	Rate	0.01	at	31.88
refire at	31.88				
OUT Size	34	Rate	0.01	at	32.88
refire at	32.88				
OUT Size	35	Rate	0.01	at	33.88
refire at	33.88				
OUT Size	36	Rate	0.01	at	34.88
refire at	34.88				
OUT Size	37	Rate	0.01	at	35.88
refire at	35.88				
OUT Size	38	Rate	0.01	at	36.88
refire at	36.88				
OUT Size	39	Rate	0.01	at	37.88
refire at	37.88				
OUT Size	40	Rate	0.01	at	38.88
refire at	38.88				
OUT Size	41	Rate	0.01	at	39.88
refire at	39.88				
OUT Size	42	Rate	0.01	at	40.88
refire at	40.88				
OUT Size	43	Rate	0.01	at	41.88
refire at	41.88				
OUT Size	44	Rate	0.01	at	42.88
refire at	42.88				
OUT Size	45	Rate	0.01	at	43.88
refire at	43.88				
OUT Size	46	Rate	0.01	at	44.88
refire at	44.88				
OUT Size	47	Rate	0.01	at	45.88
refire at	45.88				
OUT Size	48	Rate	0.01	at	46.88
refire at	46.88				
OUT Size	49	Rate	0.01	at	47.88
refire at	47.88				
OUT Size	50	Rate	0.01	at	48.88
refire at	48.88				
OUT Size	51	Rate	0.01	at	49.88
refire at	49.88				

Cuadro 5.1. *Resultados de archivo que determinan el estado actual dentro de la estrella SlidingWin para el ejemplo en cuestión. El nombre y ruta del archivo es /root/uno.ascii.*

```
SEND      0   NUM   1   at    0
&&&&ACK   0   cwnd  1   WIN   0   RETRANSMISSION 0   at  2.22
M         2.22  RTO   9
SEND      1   NUM   2   at    2.22
&&&&ACK   1   cwnd  2   WIN   2   RETRANSMISSION 0   at  4.44
M         2.22  RTO   8
SEND      2   NUM   3   at    4.44
SEND      3   NUM   4   at    4.45
&&&&ACK   2   cwnd  3   WIN   3   RETRANSMISSION 0   at  6.66
M         2.22  RTO   7.5
SEND      4   NUM   5   at    6.66
SEND      5   NUM   6   at    6.67
&&&&ACK   3   cwnd  4   WIN   4   RETRANSMISSION 0   at  7.66
SEND      6   NUM   7   at    7.66
SEND      7   NUM   8   at    7.67
&&&&ACK   4   cwnd  5   WIN   5   RETRANSMISSION 0   at  8.88
M         2.22  RTO   6.5
SEND      8   NUM   9   at    8.88
&&&&ACK   4   cwnd  5   WIN   5   RETRANSMISSION 0   at  9.88
SEND      9   NUM   10  at    9.88
&&&&ACK   4   cwnd  5   WIN   5   RETRANSMISSION 0   at  10.88
SEND      10  NUM   11  at    10.88
&&&&ACK   4   cwnd  5   WIN   5   RETRANSMISSION 0   at  11.88
SEND      11  NUM   12  at    11.88
&&&&ACK   4   cwnd  5   WIN   5   RETRANSMISSION 0   at  12.88
M         4   RTO   7
SEND      12  NUM   13  at    12.88
&&&&ACK   9   cwnd  5   WIN   5   RETRANSMISSION 0   at  13.88
SEND      13  NUM   14  at    13.88
&&&&ACK   10  cwnd  5   WIN   5   RETRANSMISSION 0   at  14.88
SEND      14  NUM   15  at    14.88
&&&&ACK   11  cwnd  5   WIN   5   RETRANSMISSION 0   at  15.88
SEND      15  NUM   16  at    15.88
&&&&ACK   12  cwnd  5   WIN   5   RETRANSMISSION 0   at  16.88
M         16  RTO   7
SEND      16  NUM   17  at    16.88
&&&&ACK   13  cwnd  5   WIN   5   RETRANSMISSION 0   at  17.88
SEND      17  NUM   18  at    17.88
&&&&ACK   14  cwnd  5   WIN   5   RETRANSMISSION 0   at  18.88
SEND      18  NUM   19  at    18.88
&&&&ACK   15  cwnd  5   WIN   5   RETRANSMISSION 0   at  19.88
SEND      19  NUM   20  at    19.88
&&&&ACK   16  cwnd  5   WIN   5   RETRANSMISSION 0   at  20.88
M         4   RTO   7
SEND      20  NUM   21  at    20.88
&&&&ACK   17  cwnd  5   WIN   5   RETRANSMISSION 0   at  21.88
```

```
SEND      21  NUM  22  at   21.88
&&&&ACK   18  cwnd  5  WIN   5   RETRANSMISSION  0   at  22.88
SEND      22  NUM  23  at   22.88
&&&&ACK   19  cwnd  5  WIN   5   RETRANSMISSION  0   at  23.88
SEND      23  NUM  24  at   23.88
&&&&ACK   20  cwnd  5  WIN   5   RETRANSMISSION  0   at  24.88
M         25  RTO  7
SEND      24  NUM  25  at   24.88
&&&&ACK   21  cwnd  5  WIN   5   RETRANSMISSION  0   at  25.88
SEND      25  NUM  26  at   25.88
&&&&ACK   22  cwnd  5  WIN   5   RETRANSMISSION  0   at  26.88
SEND      26  NUM  27  at   26.88
&&&&ACK   23  cwnd  5  WIN   5   RETRANSMISSION  0   at  27.88
SEND      27  NUM  28  at   27.88
&&&&ACK   24  cwnd  5  WIN   5   RETRANSMISSION  0   at  28.88
M         4   RTO  7
SEND      28  NUM  29  at   28.88
&&&&ACK   25  cwnd  5  WIN   5   RETRANSMISSION  0   at  29.88
SEND      29  NUM  30  at   29.88
&&&&ACK   26  cwnd  5  WIN   5   RETRANSMISSION  0   at  30.88
SEND      30  NUM  31  at   30.88
&&&&ACK   27  cwnd  5  WIN   5   RETRANSMISSION  0   at  31.88
SEND      31  NUM  32  at   31.88
&&&&ACK   28  cwnd  5  WIN   5   RETRANSMISSION  0   at  32.88
M         4   RTO  7
SEND      32  NUM  33  at   32.88
&&&&ACK   29  cwnd  5  WIN   5   RETRANSMISSION  0   at  33.88
SEND      33  NUM  34  at   33.88
&&&&ACK   30  cwnd  5  WIN   5   RETRANSMISSION  0   at  34.88
SEND      34  NUM  35  at   34.88
&&&&ACK   31  cwnd  5  WIN   5   RETRANSMISSION  0   at  35.88
SEND      35  NUM  36  at   35.88
&&&&ACK   32  cwnd  5  WIN   5   RETRANSMISSION  0   at  36.88
M         4   RTO  7
SEND      36  NUM  37  at   36.88
&&&&ACK   33  cwnd  5  WIN   5   RETRANSMISSION  0   at  37.88
SEND      37  NUM  38  at   37.88
&&&&ACK   34  cwnd  5  WIN   5   RETRANSMISSION  0   at  38.88
SEND      38  NUM  39  at   38.88
&&&&ACK   35  cwnd  5  WIN   5   RETRANSMISSION  0   at  39.88
SEND      39  NUM  40  at   39.88
&&&&ACK   36  cwnd  5  WIN   5   RETRANSMISSION  0   at  40.88
M         4   RTO  7
SEND      40  NUM  41  at   40.88
&&&&ACK   37  cwnd  5  WIN   5   RETRANSMISSION  0   at  41.88
SEND      41  NUM  42  at   41.88
```

&&&&ACK	38	cwnd	5	WIN	5	RETRANSMISSION	0	at	42.88
SEND	42	NUM	43	at	42.88				
&&&&ACK	39	cwnd	5	WIN	5	RETRANSMISSION	0	at	43.88
SEND	43	NUM	44	at	43.88				
&&&&ACK	40	cwnd	5	WIN	5	RETRANSMISSION	0	at	44.88
M	4	RTO	7						
SEND	44	NUM	45	at	44.88				
&&&&ACK	41	cwnd	5	WIN	5	RETRANSMISSION	0	at	45.88
SEND	45	NUM	46	at	45.88				
&&&&ACK	42	cwnd	5	WIN	5	RETRANSMISSION	0	at	46.88
SEND	46	NUM	47	at	46.88				
&&&&ACK	43	cwnd	5	WIN	5	RETRANSMISSION	0	at	47.88
SEND	47	NUM	48	at	47.88				
&&&&ACK	44	cwnd	5	WIN	5	RETRANSMISSION	0	at	48.88
M	4	RTO	7						
SEND	48	NUM	49	at	48.88				
&&&&ACK	45	cwnd	5	WIN	5	RETRANSMISSION	0	at	49.88
SEND	49	NUM	50	at	49.88				

Cuadro 5.2. *Resultados de archivo que determinan el estado actual dentro de la estrella TCPSender para el ejemplo en cuestión. El nombre y ruta del archivo es /root/dos.ascii.*

SEQ	0	NUM	1	at	1.12
SEQ	1	NUM	2	at	3.34
SEQ	2	NUM	3	at	5.56
SEQ	3	NUM	4	at	6.56
SEQ	4	NUM	5	at	7.78
SEQ	5	NUM	6	at	8.78
SEQ	6	NUM	7	at	9.78
SEQ	7	NUM	8	at	10.78
SEQ	8	NUM	9	at	11.78
SEQ	9	NUM	10	at	12.78
SEQ	10	NUM	11	at	13.78
SEQ	11	NUM	12	at	14.78
SEQ	12	NUM	13	at	15.78
SEQ	13	NUM	14	at	16.78
SEQ	14	NUM	15	at	17.78
SEQ	15	NUM	16	at	18.78
SEQ	16	NUM	17	at	19.78
SEQ	17	NUM	18	at	20.78
SEQ	18	NUM	19	at	21.78
SEQ	19	NUM	20	at	22.78
SEQ	20	NUM	21	at	23.78
SEQ	21	NUM	22	at	24.78

SEQ	22	NUM	23	at	25.78
SEQ	23	NUM	24	at	26.78
SEQ	24	NUM	25	at	27.78
SEQ	25	NUM	26	at	28.78
SEQ	26	NUM	27	at	29.78
SEQ	27	NUM	28	at	30.78
SEQ	28	NUM	29	at	31.78
SEQ	29	NUM	30	at	32.78
SEQ	30	NUM	31	at	33.78
SEQ	31	NUM	32	at	34.78
SEQ	32	NUM	33	at	35.78
SEQ	33	NUM	34	at	36.78
SEQ	34	NUM	35	at	37.78
SEQ	35	NUM	36	at	38.78
SEQ	36	NUM	37	at	39.78
SEQ	37	NUM	38	at	40.78
SEQ	38	NUM	39	at	41.78
SEQ	39	NUM	40	at	42.78
SEQ	40	NUM	41	at	43.78
SEQ	41	NUM	42	at	44.78
SEQ	42	NUM	43	at	45.78
SEQ	43	NUM	44	at	46.78
SEQ	44	NUM	45	at	47.78
SEQ	45	NUM	46	at	48.78
SEQ	46	NUM	47	at	49.78

Cuadro 5.3. *Resultados de archivo que determinan el estado actual dentro de la estrella TCP_Rec para el ejemplo en cuestión. El nombre y ruta del archivo es /root/tres.ascii.*

(port#1)	at time	1.12	value>	0
(port#1)	at time	3.34	value>	1
(port#1)	at time	5.56	value>	2
(port#1)	at time	6.56	value>	3
(port#1)	at time	7.78	value>	4
(port#1)	at time	8.78	value>	5
(port#1)	at time	9.78	value>	6
(port#1)	at time	10.78	value>	7
(port#1)	at time	11.78	value>	8
(port#1)	at time	12.78	value>	9
(port#1)	at time	13.78	value>	10
(port#1)	at time	14.78	value>	11
(port#1)	at time	15.78	value>	12
(port#1)	at time	16.78	value>	13
(port#1)	at time	17.78	value>	14

(port#1)	at time	18.78	value>	15
(port#1)	at time	19.78	value>	16
(port#1)	at time	20.78	value>	17
(port#1)	at time	21.78	value>	18
(port#1)	at time	22.78	value>	19
(port#1)	at time	23.78	value>	20
(port#1)	at time	24.78	value>	21
(port#1)	at time	25.78	value>	22
(port#1)	at time	26.78	value>	23
(port#1)	at time	27.78	value>	24
(port#1)	at time	28.78	value>	25
(port#1)	at time	29.78	value>	26
(port#1)	at time	30.78	value>	27
(port#1)	at time	31.78	value>	28
(port#1)	at time	32.78	value>	29
(port#1)	at time	33.78	value>	30
(port#1)	at time	34.78	value>	31
(port#1)	at time	35.78	value>	32
(port#1)	at time	36.78	value>	33
(port#1)	at time	37.78	value>	34
(port#1)	at time	38.78	value>	35
(port#1)	at time	39.78	value>	36
(port#1)	at time	40.78	value>	37
(port#1)	at time	41.78	value>	38
(port#1)	at time	42.78	value>	39
(port#1)	at time	43.78	value>	40
(port#1)	at time	44.78	value>	41
(port#1)	at time	45.78	value>	42
(port#1)	at time	46.78	value>	43
(port#1)	at time	47.78	value>	44
(port#1)	at time	48.78	value>	45
(port#1)	at time	49.78	value>	46

Cuadro 5.4. *Resultados de archivo que determinan el estado actual dentro de la estrella Printer para el ejemplo en cuestión. El nombre y ruta del archivo es /root/cuatro.ascii.*

Como puede apreciarse, la secuencia de paquetes asociada a cada uno de ellos, los números de acuse recibo, y los tamaños de localidades son evidentes a nivel gráfico y dentro de los archivos de datos generados. Con esto puede determinarse el desempeño normal de un flujo de paquetes a través del sistema de simulación.

5.3 RESULTADOS CON RETRANSMISIÓN

La retransmisión mediante el algoritmo de arranque lento puede lograrse al variar el retardo que existe en la estrella *TCPSender*, y al variar los tamaños de ventana en la estrella *SlidinWin* o bien al incrementar la ventana deslizante por el lado del receptor. Con ello conseguimos que la cwnd incremente el flujo de paquetes hasta un valor tal que genere una saturación y un retardo en los acuses de recibo. Es entonces cuando se produce una retransmisión. Variar el tamaño de la ventana en el búfer del receptor a un valor de 10, se consigue que los acuses de recibo demoren mas y se produzca retransmisión.

Por lo tanto, para poder generar una secuencia de flujo de octetos en la que se produzca una retransmisión de paquetes hasta un determinado instante del proceso se puede probar con un ejemplo utilizando el siguiente conjunto de parámetros:

PARA EL TRANSMISOR

MakeTCPPacket

Start_value	= 0
source	= 0
destination	= 0
VC	= 0

SlidingWin

capacity	= 10
MSS	= 1
Rate	= 0.01
filename	= /root/uno.ascii
Delay	= 0.01

TCPSender

Win_Size	= 5
MSS	= 1
Init_time	= 1
Rate	= 0.01
Grain	= 2

Otras Estrellas

FIFOQUEUE:
Capacity = -1
numDemandPending = 1
ConsolidateDemands = TRUE
SERVER:

Debug = 1
Filename = /root/dos.ascii
Delay = 1

ServiceTime = 0.1
%DELAY:
Delay = 0.1

PARA EL RECEPTOR

TCP_Rec

Window_Size = 10
Filename = /root/tres.ascii
MSS = 1
Ack_Time = 1
Debug = 1

ReceiberWin

Capacity = -1
MSS = 1
Delay = 0.01

Otra Estrellas

DELAY:
Delay = 0.1
%DELAY:
delay = 1
PRINTER:
Filename = /root/cuatro.ascii

PARA EL CANAL DE TRANSMISIÓN (red)

Otras Estrellas

CLOCK:
Interval = 0.01
Magnitude = 1

SERVER:
Service_Time = 1

FIFOQUEUE:
Capacity = -1
NumDemandsPending = 1
ConsolidateDemands = TRUE

DELAY (#1):
delay = 0.01

DELAY (#2):
delay = 0.01

DELAY (#3):
delay = 1

Utilizando un proceso de ejecución en Ptolemy para 50 puntos de iteración, los resultados gráficos obtenidos en las estrellas de sumidero (XMgraph) que están ubicadas a lo largo el diagrama del circuito que se presentó en las figuras 4.1, 4.5 y 4.8 para el

transmisor, receptor y la red respectivamente, se ilustran a continuación en las figuras 5.4, 5.5 y 5.6 respectivamente:

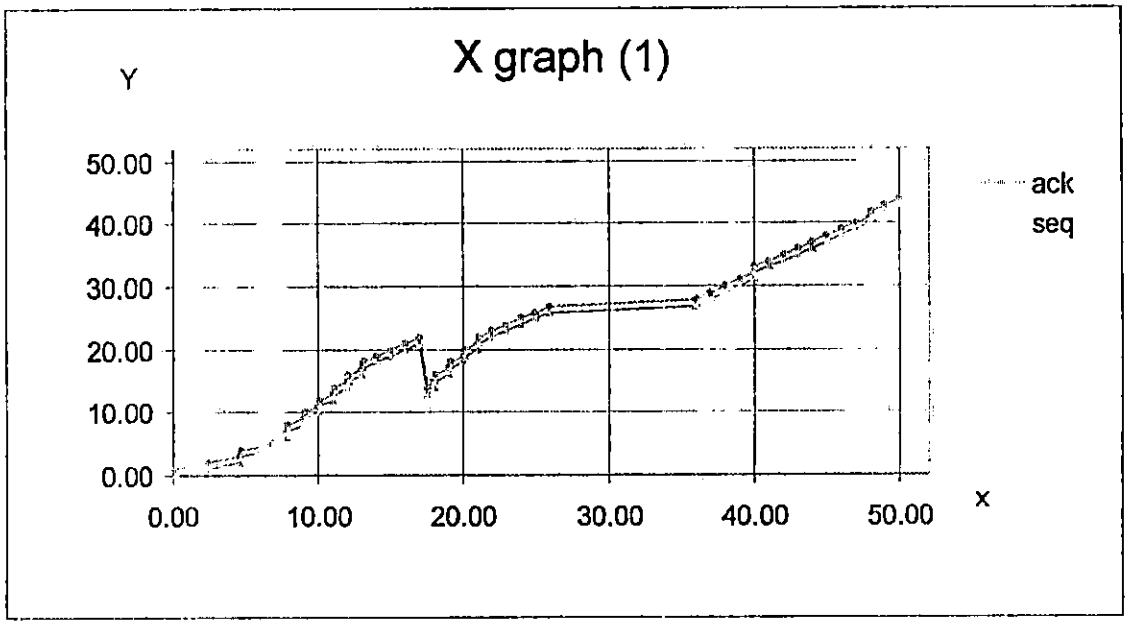


Figura 5.4. *Resultados gráficos a la salida del transmisor TCP y que representan al flujo de paquetes enviados hacia la red con destino al receptor, correspondientes a XMgraph (1) según el diagrama del circuito.*

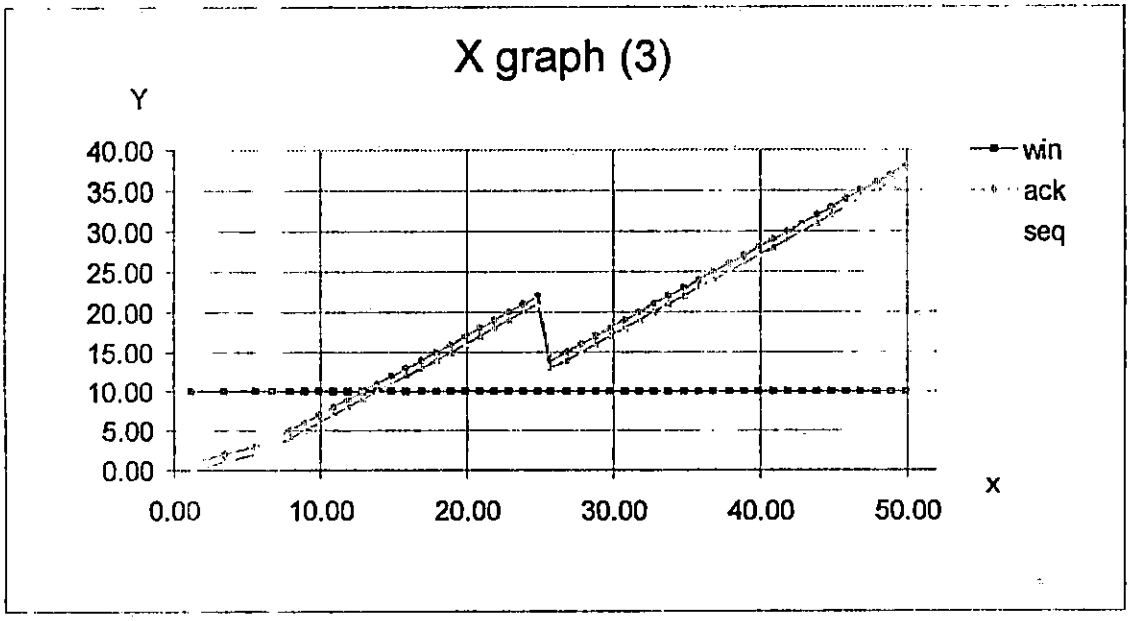


Figura 5.5. *Resultados gráficos a la salida del receptor TCP y que representan el flujo de los acuses de recibo hacia el transmisor, correspondientes a XMgraph (3) según el diagrama del circuito.*

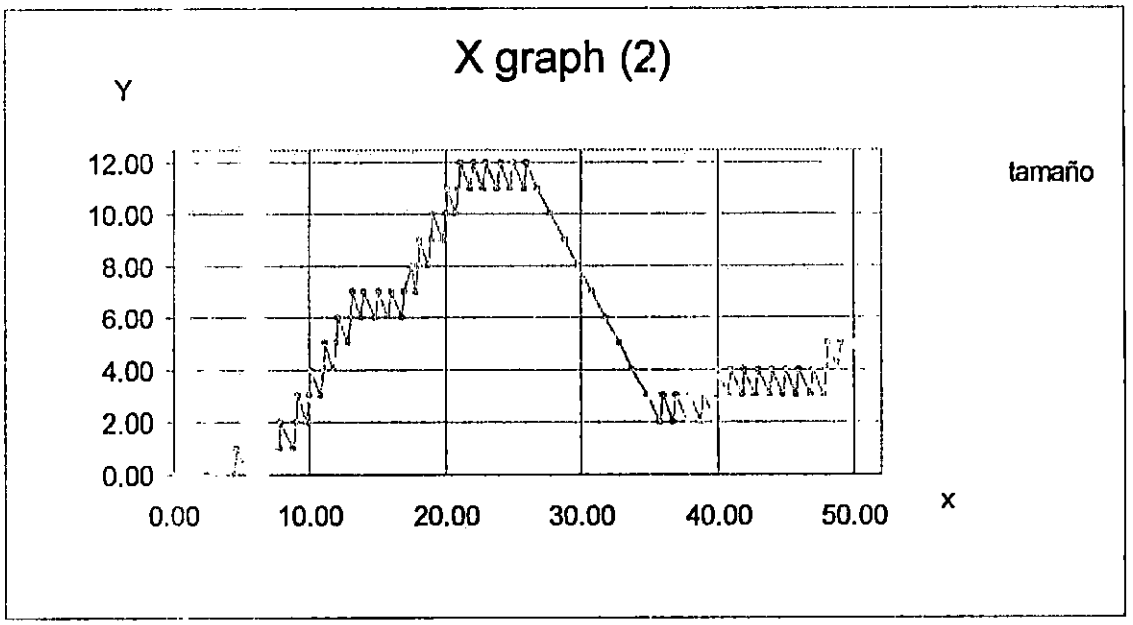


Figura 5.6. *Resultados gráficos que determinan el tamaño de la cola FIFO dentro del modelo de red y que representan el flujo de paquetes a través de ella, correspondientes a XMgraph (2) según el diagrama del circuito.*

Nuevamente, en las gráficas anteriores puede observarse claramente el tráfico de los paquetes a través del sistema, proporcionando información continua del tamaño de ventana (win), acuses de recibo (ack), números de secuencia (seq) y en la red el tamaño (size) de la cola.

El flujo de paquetes se transporta normalmente hasta el número 22, a partir de ahí se produce una expiración en el tiempo, y se retransmiten los paquetes desde el 13 hasta el 22, luego el flujo normal se reanuda.

Esta y otra información adicional puede también apreciarse en los archivos generados por las estrellas *SlidingWin, TCPSender, TCP_Rec y Printer* y que fueron almacenados según la ruta asignada en los parámetros que se describieron arriba con los nombres /root/uno.ascii, /root/dos.ascii, /root/tres.ascii y /root/cuatro.ascii respectivamente. Esta información adicional para este ejemplo con retransmisión se presenta a continuación en los cuadros 5.5, 5.6, 5.7 y 5.8:

```
NEW RESTART
refire at    2.22
OUT Size  5     Rate  0.01  at  2.22
refire at    4.44
OUT Size  5     Rate  0.01  at  4.44
OUT Size  5     Rate  0.01  at  4.45
refire at    6.66
OUT Size  7     Rate  0.01  at  6.66
OUT Size  7     Rate  0.01  at  6.67
refire at    7.66
OUT Size  9     Rate  0.01  at  7.66
OUT Size  9     Rate  0.01  at  7.67
refire at    8.88
OUT Size  11    Rate  0.01  at  8.88
OUT Size  11    Rate  0.01  at  8.89
refire at    9.88
OUT Size  13    Rate  0.01  at  9.88
OUT Size  13    Rate  0.01  at  9.89
refire at    10.88
OUT Size  15    Rate  0.01  at  10.88
OUT Size  15    Rate  0.01  at  10.89
refire at    11.88
OUT Size  17    Rate  0.01  at  11.88
OUT Size  17    Rate  0.01  at  11.89
refire at    12.88
OUT Size  19    Rate  0.01  at  12.88
OUT Size  19    Rate  0.01  at  12.89
refire at    13.88
OUT Size  20    Rate  0.01  at  13.88
refire at    14.88
OUT Size  21    Rate  0.01  at  14.88
refire at    15.88
OUT Size  22    Rate  0.01  at  15.88
refire at    16.88
OUT Size  23    Rate  0.01  at  16.88
refire at    17.39
refire at    17.88
refire at    18.88
refire at    19.88
refire at    20.88
refire at    21.88
OUT Size  23    Rate  0.01  at  21.88
refire at    22.88
OUT Size  24    Rate  0.01  at  22.88
refire at    23.88
OUT Size  25    Rate  0.01  at  23.88
```

refire at	24.88				
OUT Size	26	Rate	0.01	at	24.88
refire at	25.88				
OUT Size	28	Rate	0.01	at	25.88
refire at	35.88				
OUT Size	29	Rate	0.01	at	35.88
refire at	36.88				
OUT Size	30	Rate	0.01	at	36.88
refire at	37.88				
OUT Size	31	Rate	0.01	at	37.88
refire at	38.88				
OUT Size	32	Rate	0.01	at	38.88
refire at	39.88				
OUT Size	34	Rate	0.01	at	39.88
OUT Size	34	Rate	0.01	at	39.89
refire at	40.88				
OUT Size	35	Rate	0.01	at	40.88
refire at	41.88				
OUT Size	36	Rate	0.01	at	41.88
refire at	42.88				
OUT Size	37	Rate	0.01	at	42.88
refire at	43.88				
OUT Size	38	Rate	0.01	at	43.88
refire at	44.88				
OUT Size	39	Rate	0.01	at	44.88
refire at	45.88				
OUT Size	40	Rate	0.01	at	45.88
refire at	46.88				
OUT Size	41	Rate	0.01	at	46.88
refire at	47.88				
OUT Size	43	Rate	0.01	at	47.88
OUT Size	43	Rate	0.01	at	47.89
refire at	48.88				
OUT Size	44	Rate	0.01	at	48.88
refire at	49.88				
OUT Size	45	Rate	0.01	at	49.88

Cuadro 5.5. *Resultados de archivo que determinan el estado actual dentro de la estrella SlidingWin para el ejemplo en cuestión. El nombre y ruta del archivo es /root/uno.ascii.*

```
SEND      0    NUM    1    at     0
&&&&ACK   0    cwnd   1    WIN    0    RETRANSMISSION  0    at  2.22
M         2.22 RTO    9
SEND      1    NUM    2    at     2.22
&&&&ACK   1    cwnd   2    WIN    2    RETRANSMISSION  0    at  4.44
M         2.22 RTO    8
SEND      2    NUM    3    at     4.44
SEND      3    NUM    4    at     4.45
&&&&ACK   2    cwnd   3    WIN    3    RETRANSMISSION  0    at  6.66
M         2.22 RTO    7.5
SEND      4    NUM    5    at     6.66
SEND      5    NUM    6    at     6.67
&&&&ACK   3    cwnd   4    WIN    4    RETRANSMISSION  0    at  7.66
SEND      6    NUM    7    at     7.66
SEND      7    NUM    8    at     7.67
&&&&ACK   4    cwnd   5    WIN    5    RETRANSMISSION  0    at  8.88
M         2.22 RTO    6.5
SEND      8    NUM    9    at     8.88
SEND      9    NUM    10   at     9.89
&&&&ACK   5    cwnd   6    WIN    6    RETRANSMISSION  0    at  9.88
SEND      10   NUM    11   at     9.88
SEND      11   NUM    12   at     9.89
&&&&ACK   6    cwnd   7    WIN    7    RETRANSMISSION  0    at  10.88
SEND      12   NUM    13   at     10.88
SEND      13   NUM    14   at     10.89
&&&&ACK   7    cwnd   8    WIN    8    RETRANSMISSION  0    at  11.88
SEND      14   NUM    15   at     11.88
SEND      15   NUM    16   at     11.89
&&&&ACK   8    cwnd   9    WIN    9    RETRANSMISSION  0    at  12.88
M         4    RTO    7
SEND      16   NUM    17   at     12.88
SEND      17   NUM    18   at     12.89
&&&&ACK   9    cwnd   10   WIN    10   RETRANSMISSION  0    at  13.88
SEND      18   NUM    19   at     13.88
&&&&ACK   10   cwnd   10   WIN    10   RETRANSMISSION  0    at  14.88
SEND      19   NUM    20   at     14.88
&&&&ACK   11   cwnd   10   WIN    10   RETRANSMISSION  0    at  15.88
SEND      20   NUM    21   at     15.88
&&&&ACK   12   cwnd   10   WIN    10   RETRANSMISSION  0    at  16.88
SEND      21   NUM    22   at     16.88
TIMEOUT   13   Resend 13   NUM    14   at 17.39
&&&&ACK   13   cwnd   1    WIN    10   RETRANSMISSION  4    at  17.88
Ack 14    Last_Sent    22
RESEND    14   last_RET 14
RESEND    15   last_RET 15
&&&&ACK   14   cwnd   2    WIN    2    RETRANSMISSION  4    at  18.88
```

```
Ack 15   Last_Sent    22
RESEND   16    last_RET  16
RESEND   17    last_RET  17
&&&&ACK  15   cwnd   3  WIN   3   RETRANSMISSION  4   at  19.88
Ack 16   Last_Sent    22
RESEND   18    last_RET  18
RESEND   19    last_RET  19
&&&&ACK  16   cwnd   4  WIN   4   RETRANSMISSION  4   at  20.88
Ack 17   Last_Sent    22
RESEND   20    last_RET  20
RESEND   21    last_RET  21
&&&&ACK  17   cwnd  5.2 WIN  5.2  RETRANSMISSION  4   at  21.88
SEND     22   NUM   23   at  21.88
&&&&ACK  18   cwnd  5.4 WIN 5.392 RETRANSMISSION  0   at  22.88
SEND     23   NUM   24   at  22.88
&&&&ACK  19   cwnd  5.6 WIN 5.578 RETRANSMISSION  0   at  23.88
SEND     24   NUM   25   at  23.88
&&&&ACK  20   cwnd  5.8 WIN 5.757 RETRANSMISSION  0   at  24.88
SEND     25   NUM   26   at  24.88
&&&&ACK  21   cwnd  5.9 WIN 5.931 RETRANSMISSION  0   at  25.88
SEND     26   NUM   27   at  25.88
&&&&ACK  13   cwnd  6.1 WIN 6.099 RETRANSMISSION  0   at  26.78
&&&&ACK  14   cwnd  6.1 WIN 6.099 RETRANSMISSION  0   at  27.78
&&&&ACK  15   cwnd  6.1 WIN 6.099 RETRANSMISSION  0   at  28.78
&&&&ACK  16   cwnd  6.1 WIN 6.099 RETRANSMISSION  0   at  29.78
&&&&ACK  17   cwnd  6.1 WIN 6.099 RETRANSMISSION  0   at  30.78
&&&&ACK  18   cwnd  6.1 WIN 6.099 RETRANSMISSION  0   at  31.78
&&&&ACK  19   cwnd  6.1 WIN 6.099 RETRANSMISSION  0   at  32.78
&&&&ACK  20   cwnd  6.1 WIN 6.099 RETRANSMISSION  0   at  33.78
&&&&ACK  21   cwnd  6.1 WIN 6.099 RETRANSMISSION  0   at  34.78
&&&&ACK  22   cwnd  6.3 WIN 6.263 RETRANSMISSION  0   at  35.88
M        14   RTO   19
SEND     27   NUM   28   at  35.88
&&&&ACK  23   cwnd  6.4 WIN 6.423 RETRANSMISSION  0   at  36.88
SEND     28   NUM   29   at  36.88
&&&&ACK  24   cwnd  6.6 WIN 6.579 RETRANSMISSION  0   at  37.88
SEND     29   NUM   30   at  37.88
&&&&ACK  25   cwnd  6.7 WIN 6.731 RETRANSMISSION  0   at  38.88
SEND     30   NUM   31   at  38.88
&&&&ACK  26   cwnd  6.9 WIN 6.879 RETRANSMISSION  0   at  39.88
SEND     31   NUM   32   at  39.88
SEND     32   NUM   33   at  39.89
&&&&ACK  27   cwnd   7  WIN 7.025 RETRANSMISSION  0   at  40.88
M         5   RTO   16
SEND     33   NUM   34   at  40.88
&&&&ACK  28   cwnd  7.2 WIN 7.167 RETRANSMISSION  0   at  41.88
```

SEND	34	NUM	35	at	41.88				
&&&&ACK	29	cwnd	7.3	WIN	7.306	RETRANSMISSION	0	at	42.88
SEND	35	NUM	36	at	42.88				
&&&&ACK	30	cwnd	7.4	WIN	7.443	RETRANSMISSION	0	at	43.88
SEND	36	NUM	37	at	43.88				
&&&&ACK	31	cwnd	7.6	WIN	7.578	RETRANSMISSION	0	at	44.88
SEND	37	NUM	38	at	44.88				
&&&&ACK	32	cwnd	7.7	WIN	7.71	RETRANSMISSION	0	at	45.88
SEND	38	NUM	39	at	45.88				
&&&&ACK	33	cwnd	7.8	WIN	7.839	RETRANSMISSION	0	at	46.88
M	6	RTO	15						
SEND	39	NUM	40	at	46.88				
&&&&ACK	34	cwnd	8	WIN	7.967	RETRANSMISSION	0	at	47.88
SEND	40	NUM	41	at	47.88				
SEND	41	NUM	42	at	47.89				
&&&&ACK	35	cwnd	8.1	WIN	8.092	RETRANSMISSION	0	at	48.88
SEND	42	NUM	43	at	48.88				
&&&&ACK	36	cwnd	8.2	WIN	8.216	RETRANSMISSION	0	at	49.88
SEND	43	NUM	44	at	49.88				

Cuadro 5.6. *Resultados de archivo que determinan el estado actual dentro de la estrella TCPSender para el ejemplo en cuestión. El nombre y ruta del archivo es /root/dos.ascii.*

SEQ	0	NUM	1	at	1.12
SEQ	1	NUM	2	at	3.34
SEQ	2	NUM	3	at	5.56
SEQ	3	NUM	4	at	6.56
SEQ	4	NUM	5	at	7.78
SEQ	5	NUM	6	at	8.78
SEQ	6	NUM	7	at	9.78
SEQ	7	NUM	8	at	10.78
SEQ	8	NUM	9	at	11.78
SEQ	9	NUM	10	at	12.78
SEQ	10	NUM	11	at	13.78
SEQ	11	NUM	12	at	14.78
SEQ	12	NUM	13	at	15.78
SEQ	13	NUM	14	at	16.78
SEQ	14	NUM	15	at	17.78
SEQ	15	NUM	16	at	18.78
SEQ	16	NUM	17	at	19.78
SEQ	17	NUM	18	at	20.78
SEQ	18	NUM	19	at	21.78
SEQ	19	NUM	20	at	22.78
SEQ	20	NUM	21	at	23.78

SEQ	21	NUM	22	at	24.78
SEQ	13	NUM	14	at	25.78
SEQ	14	NUM	15	at	26.78
SEQ	15	NUM	16	at	27.78
SEQ	16	NUM	17	at	28.78
SEQ	17	NUM	18	at	29.78
SEQ	18	NUM	19	at	30.78
SEQ	19	NUM	20	at	31.78
SEQ	20	NUM	21	at	32.78
SEQ	21	NUM	22	at	33.78
SEQ	22	NUM	23	at	34.78
SEQ	23	NUM	24	at	35.78
SEQ	24	NUM	25	at	36.78
SEQ	25	NUM	26	at	37.78
SEQ	26	NUM	27	at	38.78
SEQ	27	NUM	28	at	39.78
SEQ	28	NUM	29	at	40.78
SEQ	29	NUM	30	at	41.78
SEQ	30	NUM	31	at	42.78
SEQ	31	NUM	32	at	43.78
SEQ	32	NUM	33	at	44.78
SEQ	33	NUM	34	at	45.78
SEQ	34	NUM	35	at	46.78
SEQ	35	NUM	36	at	47.78
SEQ	36	NUM	37	at	48.78
SEQ	37	NUM	38	at	49.78

Cuadro 5.7. *Resultados de archivo que determinan el estado actual dentro de la estrella TCP_Rec para el ejemplo en cuestión. El nombre y ruta del archivo es /root/tres.ascii.*

(port#1)	at time	1.12	value>	0
(port#1)	at time	3.34	value>	1
(port#1)	at time	5.56	value>	2
(port#1)	at time	6.56	value>	3
(port#1)	at time	7.78	value>	4
(port#1)	at time	8.78	value>	5
(port#1)	at time	9.78	value>	6
(port#1)	at time	10.78	value>	7
(port#1)	at time	11.78	value>	8
(port#1)	at time	12.78	value>	9
(port#1)	at time	13.78	value>	10
(port#1)	at time	14.78	value>	11
(port#1)	at time	15.78	value>	12
(port#1)	at time	16.78	value>	13

(port#1)	at time	17.78	value>	14
(port#1)	at time	18.78	value>	15
(port#1)	at time	19.78	value>	16
(port#1)	at time	20.78	value>	17
(port#1)	at time	21.78	value>	18
(port#1)	at time	22.78	value>	19
(port#1)	at time	23.78	value>	20
(port#1)	at time	24.78	value>	21
(port#1)	at time	34.78	value>	22
(port#1)	at time	35.78	value>	23
(port#1)	at time	36.78	value>	24
(port#1)	at time	37.78	value>	25
(port#1)	at time	38.78	value>	26
(port#1)	at time	39.78	value>	27
(port#1)	at time	40.78	value>	28
(port#1)	at time	41.78	value>	29
(port#1)	at time	42.78	value>	30
(port#1)	at time	43.78	value>	31
(port#1)	at time	44.78	value>	32
(port#1)	at time	45.78	value>	33
(port#1)	at time	46.78	value>	34
(port#1)	at time	47.78	value>	35
(port#1)	at time	48.78	value>	36
(port#1)	at time	49.78	value>	37

Cuadro 5.8. *Resultados de archivo que determinan el estado actual dentro de la estrella Printer para el ejemplo en cuestión. El nombre y ruta del archivo es /root/cuatro.ascii.*

Tal como puede apreciarse del archivo generado por la *TCPSender*, en el cuadro 5.6, al final del proceso aparecen tres ack's duplicados que fueron enviados por el receptor indicando la detección de una secuencia incorrecta en el flujo de los paquetes. Efectivamente, al limitar la ventana de la *SlidingWin* a 10, una vez enviados los diez paquetes, ya no hay otros mas en ese instante por lo que se "dispara" el número de secuencia para los paquetes siguientes. Los acuses de recibo siguen llegando al transmisor, hasta el número diez, luego se informa el fuera de secuencia con el envío de los tres ack's acompañado también del estado de las ventanas locales, el número de secuencia del último paquete enviado y el acuse de recibo correspondiente al último paquete correcto de debió haber llegado al receptor.

5.4 CONCLUSION.

El simulador del TCP es una herramienta de análisis importante que tiene por objetivo, ayudar al estudio del control de transporte a través de una red de comunicaciones. Su interpretación suele ser muy complicada por el hecho de abordar dentro de su estructura la mayor parte de servicios mas comunes que ofrece el protocolo TCP en sí, y por ello se requiere de un entendimiento teórico previo antes de intentar utilizarlo.

Puede observarse que es posible encontrar numerosas combinaciones posibles mediante el manipuleo de sus parámetros de operación, de los cuales no todos pueden tener sentido, es decir, si en realidad se tiene interés por algún suceso específico dentro del sistema bajo condiciones particulares de operación, estas deben realizarse con el debido cuidado para obtener simulaciones adecuadas que puedan claramente describir el comportamiento apropiado, para esas condiciones específicas.

El uso del canal de transmisión juega un valioso papel para el desempeño de la comunicación entre la fuente y el destino, por lo que también debe tomarse muy en cuenta a la hora de hacer las configuraciones respectivas, con los anchos de banda adecuados.

BIBLIOGRAFÍA

[1] Ptolemy 0.7 User's Manual. "The Almagest". Vol 1 University of California Berkeley. College of Engineering. Department of Electrical Engineering and Computer Sciences.

[2] Ptolemy Programmer's Manual. "The Almagest". Vol 2. University of California Berkeley. College of Engineering. Department of Electrical Engineering and Computer Sciences.

[3] Ferrel G. Streimbler. "Introducción a los sistemas de Comunicaciones", Editorial Addison-Wesley.

[4] Hernando Rábanos José María, "Sistemas de Telecomunicación, Transmisión por línea y Redes. Segunda edición, servicio de Publicaciones E.T.S.I. de Telecomunicaciones Editorial Centro Estudio Ramón Areces, 1991.

[5] Uyless Black, "Computer Networks Protocols, Standards and Interfaces", PH de ISBN-0-13-17605-2

[6] Addisson-Wesley-Longman, "Comunicación de Datos, Redes de Computadoras y Sistemas Abiertos". KBN-968444331-5

[7] Douglas E. Comer "Redes Globales de Información con Internet y TCP/IP" Principios Básicos, Protocolos y Arquitectura, 3ª Edición, Prentice Hall Hispanoamericana, 1995.

[8] M Group, "Communications and Networking for the IBM PC & Compatibles, Third Edition.

[9] Brian W. Kernighan, Dennis M. Ritchie "El Lenguaje de Programación C". Bell Laboratories Murria Hill, New Jersey. Prentice Hall Hispanoamericana.

REFERENCIAS DE INTERNET

[1] © CESCA, IB/180100 http://www.cesca.es/esp/servicios/SOFT/soft.html

[2] Mate Linux http://aquiles.uca.es/matelinux/software.html

[3] Department of EECS, UC Berkeley http://ptolemy.eecs.berkeley.edu

[4] Ptolemy para Windows http://informatica.uv.es/guia/asignatu/AA/ptolemy/

[5] TKN Telecomunications Networks Group http://www-tkn.ee.tu-berlin.de/

[6] Departamento de Matemática Aplicada y Telemática de la UPC
http://elvis.upc.es/~labt2/PTOLEMY/ptolemy/main.htm

[7] Archivos Binarios ftp://ptolemy.eecs.berkeley.edu/pub/ptolemy/contrib

[8] A TCP Simulator With Ptolemy
http://www.fokus.gmd.de/step/simulations/sim.html#TCP

ANEXOS

ANEXO A
PROGRAMACIÓN DE LAS ESTRELLAS

1. MakeTCPPacket:

```
defstar {
      name { MakeTCPPacket }
      domain { DE }
      author { Bernd Deffner & DorghamSisalem }
        copyright { See $PTOLEMY_SIMULATIONS/TCP/copyright }

      location { $PTOLEMY_SIMULATIONS/TCP }
      desc { This star generates an empty TCP-Packet at firing.
                The Packet used for simulation is a data structure
consisting
                of the  information fields source, destination,
Acknowledgment
                Size, sequence byte number and sequence number.
                As an input it gets the size of the packet.
      }

      input { name { input } type { float } }
      output { name { output } type { message } }

      defstate {
            name { start_value }
            type { int  }
            default { 0 }
            desc { Start value for the sequence counter.}
      }

      defstate {
            name { source }
            type { int }
            default { 0}
            desc { number of source node.}
      }
      defstate {
            name { destination }
            type { int  }
            default { 0 }
            desc { number of destination node.}
      }
      defstate {
            name { VC }
            type { int  }
            default { 0 }
            desc { number of destination node.}
      }
      defstate {
            name { mode }
            type { int  }
            default { 0 }
            desc {0=ECN off, 1=ECN on.}
      }
      hinclude { "TCPPacket.h" }

      protected { long count;
                  long num;}
```

```
        setup {
                delayType= FALSE;
              count = start_value;
              num=0;
        }

        go {
                  completionTime= arrivalTime;
              int Size=input%0; //read the desired packet size
                num+=Size;
              LOG_NEW;
                  TCPPacket* Packet = new TCPPacket( count ); //create the
packet
int ack=(count+1)*Size;
                  // set source and destination,size and byte sequence
                  Packet->setSource(source);
                  Packet->setVC(VC);
                  Packet->setDestination(destination);
                    Packet->setSize(Size);
                    Packet->setAck(ack);
              Packet->setNum(num);
              Packet->setWin(20000);
              if(mode)
                      Packet->setECN_ON(1);
              else
                      Packet->setECN_ON(0);
                Packet->setECN(0);

              // send the envelope
              Envelope envlp( *Packet );
              output.put( completionTime ) << envlp;
              count++;
        } // end go
} // end defstar
```

2. SlidingWin:

```
defstar {
      name { SlidingWin }
      domain { DE }
      author { Dorgham Sisalem }
        copyright { SEE $PTOLEMY_SIMULATIONS/TCP/copyright }

        derivedfrom{RepeatStar}
        desc{
        This star receives the input Packets and sends them to the TCP
source
        if this is possible -i.e. the transmission window is not full
yet.
        Otherwise it buffers the Packets and waits till the source sends
        a notification, that the window has moved and it is possible to
send
        data. In this case it sends the allowed number of bytes.
        }

      input {
                name { refire}
              type { float }
```

```
            desc {send another packet
                  }
      }
      input {
              name { demand}
            type { int }
            desc { number of bytes that can be sent before the
transmission
                  window closes }
      }
      input {
              name { inData }
            type { anytype}
            desc { new data}
      }

      output {
            name {restart}

            type {float}
              desc { send another packet}
      }
      output {
            name {outData}
            type {=inData}
              desc { send data to the TCP source }
      }
      output {
            name {size}
            type {int}
            desc { Size of Queue. }
      }
      output {
            name {overflow}
            type {=inData}
            desc {
              Arrival data that cannot be queued due to capacity limit.
            }
      }
      protected {
            int infinite;
      }
      defstate {
            name {capacity}
            type {int}
            default {"-1"}
            desc { Maximum size of the queue. If <0, capacity is
infinite. }
      }
      defstate {
            name {MSS}
            type {int}
            default {"1"}
            desc {Maximum segment size }
      }
      defstate {
            name {Rate}
            type {float}
            default {"1"}
            desc {the sending rate of the source }
```

```
        }

constructor {
      // Indicate to the scheduler that size and outputs are
          // trigged by demand inputs.
      demand.triggers(size);
      inData.triggers(overflow);

         demand.before(inData);
         demand.before(refire);
         refire.triggers(restart);
         refire.triggers(outData);
         inData.triggers(outData);
         inData.triggers(size);
}

ccinclude {
      "DataStruct.h","TCPPacket.h"
}
protected {
      Queue queue;
}
method {
      name { enqueue }
      access { protected }
      arglist { "()" }
      type { void }
      code {
          if (infinite || Qsize() < int(capacity)) {
            queue.put(inData.get().clone());
          } else {
            // Data is lost
            overflow.put(completionTime) = inData.get();
          }
      }
}
method {
      name { dequeue }
      access { protected }
      arglist { "()" }
      type { Pointer }
      code {
          return queue.get();
      }
}
method {
      name { handleOverflow }
      access { protected }
      arglist { "()" }
      type { void }
      code {
          // Nothing to do
      }
}
method {
      name { Qsize }
      access { protected }
      arglist { "()" }
      type { int }
      code {
```

```
            return queue.length();
        }
    }
    method {
        name { zapQueue }
        code {
            while (Qsize() > 0) {
                Particle* pp = (Particle*) queue.get();
                pp->die();
            }
            queue.initialize();
            }
        }
    protected{
      int control,Size;
      float Completion_Time;
    }
    defstate {
        name { fileName }
        type { string }
        default { "<stdout>" }
        desc { Filename for output }
    }
    hinclude { "pt_fstream.h" }
    protected {
        pt_ofstream *p_out;
    }

    setup {
        infinite = (int(capacity) < 0);
        control=MSS;
        Completion_Time=-1;
        Size=MSS;
        zapQueue();
            LOG_DEL; delete p_out;p_out=NULL;
        LOG_NEW; p_out = new pt_ofstream(fileName);
        //DERepeatStar::setup();
    }

      wrapup {
            LOG_DEL; delete p_out;
          p_out = 0;
          }
    destructor { LOG_DEL; delete p_out;}
      constructor {p_out=NULL;
                delayType = TRUE;
                  }
    go {
        pt_ofstream& OUTPUT = *p_out;
       completionTime = arrivalTime;

//*************************************************************************
***
//          update the allowed number of byte to be sent
//*************************************************************************
***

        if (demand.dataNew)
        {
          demand.dataNew= FALSE;
```

```
              control=demand%0;     // update the allowed number of
                                    // packets to be sent.
              float tmp=completionTime-Completion_Time-Rate+1;
//OUTPUT<<" TMP "<<tmp;
              tmp=(float)100000000.0*tmp;
//OUTPUT<<" TMP1 "<<tmp;
              tmp=tmp-(float)100000000.0;
//OUTPUT<<" TMP2 "<<tmp<<" at "<<completionTime<<"\n";
           if(tmp>10)
           {
//OUTPUT<<" refire  at "<<completionTime<<"\n";
                 //restart.put(completionTime)<<0;    //schedule the next
send
               refire.dataNew=TRUE;
             }

           }


//*************************************************************************
***
// If there is new inData, store it if the window is full
//*************************************************************************
***

           if (inData.dataNew)
           {
             inData.dataNew=FALSE;
             if(Qsize())
             {
               enqueue();
             return;
             }
             if((Size<=control)&&
                   (completionTime>Completion_Time+Rate))
             {
//OUTPUT<<" NEW RESTART "<<"\n";
               Envelope Envp;
               inData.get().getMessage(Envp);
             outData.put(completionTime) <<Envp;
               refire.dataNew=TRUE;

                 //restart.put(completionTime)<<0;    //schedule the next
send
               const TCPPacket* Packet=(const TCPPacket *)Envp.myData();
             Size+=Packet->readSize();
               Completion_Time=completionTime;
             }
             else
             enqueue();
             size.put(completionTime)<<Qsize();
           }
//*************************************************************************
*
//   send a new packet .
//*************************************************************************
*

           if(refire.dataNew)
           {
```

```
            refire.dataNew=FALSE;
            if((Size<=control)&&Qsize())
            {
            Particle* pp =(Particle *)  dequeue();
            Envelope Envp;
            pp->getMessage(Envp);
            const TCPPacket* Packet=(const TCPPacket *)Envp.myData();
            Size+=Packet->readSize();
//OUTPUT<<" OUT Size "<<control<<" Rate "<<Rate<<" at
"<<completionTime<<" \n";
            outData.put(completionTime)<<Envp;
                restart.put(completionTime)<<0;//schedule the next send
            pp->die();
              Completion_Time=completionTime;
            }
          }

      }

        destructor {
            zapQueue();
        }
}
```

3. TCPSender:

```
defstar {
      name {TCPSender }
      domain { DE }
      author { Dorgham Sisalem}
        copyright { SEE $PTOLEMY_SIMULATIONS/tcp/copyright }

      desc {
        This star is a TCP source that is based on the 4.3Tahoe BSD TCP
        implementation and fulfills the following requirements:
        Sliding window mechanism: It only sends if the receiver has
enough
                space. Otherwise it waits until the receiver has freed
some
                buffer and reopened the transmission window.
        Timeout: If an acknowledgment for a specific packet has not
arrived
                after some time, it is resent.
        Round trip time estimation: Using the Karn's algorithm.
        Slow start: For the beginning of the transmission and after a
timeout.
        Congestion avoidance: A slow increase of the window size of
1/cwnd
        Fast retransmission: After receiving 3 duplicate acks the source
                             retransmits the lost packet.
      }

//************************************************************************
//                      Define Ports
//************************************************************************
      input { name { ack}
              type { message}
            desc { Acknowledgment packet}
```

```
        }
        input { name { input }
                  type { message}
                desc { new data packets}
          }
        input { name { timer}
                  type { message}
        }

        output { name { demand}
                   type { int}
                 desc { get demand number of bytes from the WindowBuf}
          }
        output { name { output }
                   type { message }
                 desc { send data packets}
        }
        output { name { timeout}
                   type { int}
                 desc { last calculated round trip time}
        }

//*************************************************************************
//                        Define States
//*************************************************************************

      defstate {
            name { Win_Size}
            type { int  }
            default { 10 }
            desc {maximum transmission window. }
      }
      defstate {
            name { MSS}
            type { int  }
            default { 1 }
            desc {Maximum Segment Size }
      }
      defstate {
            name { init_time}
            type { float }
            default { 1 }
            desc {initial value for RTT}
      }
      defstate {
            name { Rate}
            type { float }
            default { 1 }
            desc {the sending rate of the source}
      }
      defstate {
            name { Grain}
            type { float }
            default { 2 }
            desc {the number of clock ticks in a second}
      }
      defstate {
            name { debug}
            type { int }
            default { 0 }
```

```
        desc {0 output debug infos}
    }

    hinclude { "TCPPacket.h" ,<math.h>}

    defstate {
        name { fileName }
        type { string }
        default { "<stdout>" }
        desc { Filename for output }
    }
    hinclude { "pt_fstream.h" }
    protected {
        pt_ofstream *p_out;
    }

    protected { long count;
                long Last_Sent,counter;
                long Limit,state,Retransmission;
                long Ack;
                const int /*long*/ STOP=0;
                const int /*long*/ GO=1;
                  Envelope* buffer;
                long Last_Ack,Last_Win,Last_Limit;
                float cwnd,ssthresh,Win;
                long Slow_Start,Congestion_Avoidance,Time_Out;
                long Retransmitted,Timer,First_Time;
                float RTO,A,D,Start_Time;
                long Last_Seq,buffered,last_seq,last_retransmitted;
                long SEQ,Last_Retransmit,send_state;
                float *timer_buf;
                int ret;
              }

//*************************************************************************
******/
// initialization: cwnd=0;
//                 start with slow start
//                 RTO is set by the user
//*************************************************************************
******/

    setup { delayType                                   = FALSE;
            cwnd                                        =  MSS ;
          Limit                                       = long(cwnd-1);
          state                                       = GO;
          counter                                     = 0;
            Retransmission=ret                          = 0;
          Ack=Last_Ack=Last_Win                       = -1;
          Last_Limit                                  = 0;
          Slow_Start                                  = 1;
          Congestion_Avoidance=Time_Out               = 0;
          First_Time                                  = 1;
          Timer=Retransmitted                         = -2;
          RTO                                         = init_time;
          Last_Seq                                    = -1;
          send_state                                  = 1;
          ssthresh                          = Win_Size;
          Win_Size                                    = Win_Size/MSS;
            last_seq                        =-1;
```

```
            }

//*************************************************************************
******/
 // allocate buffer space for the local window
//*************************************************************************
******/

      begin  {          //initialize the local window
                if(buffer) {LOG_DEL; delete [Win_Size] buffer;
buffer=NULL;}
              LOG_NEW;
              buffer=new Envelope[Win_Size];
                if(timer_buf) {LOG_DEL; delete [Win_Size] timer_buf;
                               timer_buf=NULL;}
              LOG_NEW;
              timer_buf=new float[Win_Size];
                if(p_out){LOG_DEL; delete p_out;p_out=NULL;}
            LOG_NEW; p_out = new pt_ofstream(fileName);
            }

//*************************************************************************
*****/
// set scheduling priorities and initialize the memory
//*************************************************************************
*****/

      constructor {
                  buffer=NULL;
                  p_out=NULL;
                    timer_buf=NULL;
                  if(buffer){LOG_DEL; delete [Win_Size] buffer;
                               buffer=NULL;}
                    if(timer_buf) {LOG_DEL; delete [Win_Size] timer_buf;
                               timer_buf=NULL;}
                    ack.before(input);
                  ack.before(timer);
                  input.before(timer);
              }
        wrapup {
              if(p_out){LOG_DEL; delete p_out; p_out=NULL;}
            if(buffer){ LOG_DEL; delete [Win_Size] buffer;
                               buffer=NULL;}
              if(timer_buf) {LOG_DEL; delete [Win_Size] timer_buf;
                               timer_buf=NULL;}
            }
      destructor { if(p_out){LOG_DEL; delete p_out; p_out=NULL;}
            if(buffer) {LOG_DEL; delete [Win_Size] buffer;
                               buffer=NULL;}
              if(timer_buf) {LOG_DEL; delete [Win_Size] timer_buf;
                               timer_buf=NULL;}
      }
//*************************************************************************
//                    GO
//*************************************************************************
      go {
                pt_ofstream& OUTPUT = *p_out;

                completionTime= arrivalTime;
            if(ack.dataNew)  // an acknowlegment has arrived
```

```
            {
              Envelope Envp;
              ack.get().getMessage(Envp);
              const TCPPacket * packetPtr=(const TCPPacket *)
                Envp.myData();
              Ack=packetPtr->readAck();
              long Seq=packetPtr->readSeq();
if(debug)
OUTPUT<<" &&&&ACK "<<Seq<<" cwnd "<<cwnd<<" WIN "<<Win<<" RETRANSMISSION
"<<Retransmission<<" at "<<completionTime<<"\n";
//*************************************************************************
****/
// Ignore Acks for falsely retransmitted packets -i.e. packets that are
// retransmitted after a timeout even though the ack is on the way, or
// duplicate acks after the 3rd. one was received.
//*************************************************************************
****/

              if((Ack<Last_Ack)||((ret>2)&&(Ack==last_retransmitted)))
                 {
                  return;
                }
                if(Ack!=Last_Ack)
                  ret=0;
                  if(Ack>Last_Ack)
                      last_seq=packetPtr->readSeq();
             Win=packetPtr->readWin();

//*************************************************************************
****/
// measure round trip time only if there was no timeout for this sequence
// number.
//*************************************************************************
****/

              if((Timer!=-2)&&((Seq==Timer)||(Seq==Timer+1))
                       &&(Retransmitted!=Seq)&&(Retransmitted!=Seq-1))
                {

                    float M=arrivalTime-Start_Time;
if(debug)
OUTPUT<<" M "<<M;
                  M=M*(float)Grain;
                  int m_int=ceil(M);
                  M=(float)m_int;
                    M=M/(float)Grain;
                    if(First_Time)    // calculate the first RTO
                  {
                    A=M+0.5;
                    D=A/2;
                    RTO=A+(4*D);
                    First_Time=0;
                  }
                  else
                  {                 // calculate RTO
                    float  Err=M-A;
                    A=A+(0.125*Err);
                    if(Err<0)
                       Err*=-1;
                    D=D+(0.25*(Err-D));
```

```
                    RTO=A+(4*D);
                   }
                   Timer=-2;        //a new RTT measurement can start

                       //Round up the time measurements to the required
grain
                   RTO=RTO*(float)Grain;
                   m_int=ceil(RTO);
                   RTO=(float)m_int;
                     RTO=RTO/(float)Grain;
                   if(RTO<(Grain+Grain))
                         RTO+Grain+Grain;
if(debug)
OUTPUT<<" RTO "<<RTO<<"\n";
                   if(RTO>64)      //set an upper limit of 64
                      RTO=64;
                   //send latest round trip estimation

                   timeout.put(completionTime)<<RTO;
                  }

//*************************************************************************
****/
// Increase cwnd by the size of one packet for each acknowledged packet
//*************************************************************************
****/
                 if(Slow_Start&&(Win>cwnd))
               {
                   cwnd+=MSS;
                   if(cwnd<=0) cwnd=MSS;
                   if(cwnd>ssthresh) {cwnd=ssthresh; Slow_Start=0;}

               }

//*************************************************************************
****/
// Increase cwnd by the size of one packet for each RTT while in the
// congestion avoidance state.
//*************************************************************************
****/

               if(cwnd<Win)
                 {
                     if((cwnd>=ssthresh)&&Congestion_Avoidance)
                    {
                      Slow_Start=0;
                    float cwnd_pack=cwnd/(float)MSS;
                    cwnd_pack+=(1/cwnd_pack);
                      cwnd_pack*=MSS;
                    cwnd=cwnd_pack;
                    }
                    Win=cwnd;
               }

              Limit=Win+Ack;  //update the sliding window
//*************************************************************************
****/
// If 3 duplicate Acks were received then retransmit the lost packet and
go into
// slow start.
```

```
//*************************************************************************
****/
              if((Ack==Last_Ack)&&Last_Win) //duplicate Acks ?
                {
                 ret++;
                   if(ret<3)
                    return;
                   if(ret==3)
                  {
                      Retransmission=ret;
                  Timer=-2;          //stop the RTO calculation
                    send_state=-1;
                  Limit=0;
                  Retransmitted=Seq+1;

//*************************************************************************
*****
//clear all the timeout stamps in order to avoid timeout handling for
packets
//that were transmitted but can not be acknowledged until a retransmitted
//packet reaches the receiver
//*************************************************************************
*****
                  long i;
                  for(i=0;i<Win_Size;i++)
                       timer_buf[i]=-1;
                      timer_buf[(Seq+1)%Win_Size]=completionTime+RTO;
if(debug)
OUTPUT<<" (Seq+1)%Win_Size= "<<(Seq+1)%Win_Size<<" RTO
"<<completionTime+RTO<<"\n";
                    //read the lost packet and send it

output.put(completionTime)<<buffer[(Seq+1)%Win_Size];
                      last_retransmitted=Ack;
                    demand.put(completionTime)<<(int)STOP;
                    Last_Seq=Seq+1;

                  Slow_Start=1;  //go into slow start
                    Congestion_Avoidance=1; //start the congestion
                                            //avoidance state after
                                    //the slow start

                  if(cwnd<Win)    //reset the ssthresh value
                    {
                       ssthresh=cwnd/2;
                     ssthresh-=(long)ssthresh%MSS;
                    }
                    else
                    {
                       ssthresh=Win/2;
                     ssthresh-=(long)ssthresh%MSS;
                    }
                    if(ssthresh<MSS) ssthresh=MSS+MSS; // set the lower
                                                       // limit of
ssthresh
                  cwnd=MSS;           // initialize cwnd

                 }
                }
                else
```

```
                {
                    Last_Ack=Ack;
                Last_Win=Win;
                send_state=1;

                if(Retransmitted>=Seq)
                     Time_Out=0;      //reset the Time_Out state
              }
              if(Win<MSS) // avoid silly windows
              {
                state=STOP;
                demand.put(completionTime)<<(int)state;
              }
//All packets waiting for acknowledgements were achnowledged so clear the
flags
                if((Retransmission>=3)&&
                    ((last_retransmitted>=Last_Sent)||(Ack>=Last_Sent)))
                  {
                        Retransmitted=-2;
                        Retransmission=0;
                  }
//There are still some packets waiting for acks after a loss so resend
them
                if((Ack<Last_Sent)&&(Retransmission>=3))
                {
if(debug)
OUTPUT<<" Ack "<<Ack<<" Last_Sent "<<Last_Sent<<"\n";
                  if(last_retransmitted<Ack)
                        last_retransmitted=Ack;
                  while((Limit>last_retransmitted)&&
                        (last_retransmitted<Last_Sent))
                  {
                         float tmp=(float)last_retransmitted/(float)MSS;
if(debug)
OUTPUT<<" RESEND "<<tmp<<" last_RET "<<last_retransmitted<<"\n";

timer_buf[(int)tmp%Win_Size]=completionTime+RTO;

output.put(completionTime)<<buffer[(int)tmp%Win_Size];

                        last_retransmitted+=MSS;
                  }
                  if(last_retransmitted>=Last_Sent)
                  {
                        Retransmitted=-2;
                        Retransmission=0;
                  }

                  }
              if(Limit)  //update the sliding window size
              {
                 demand.put(completionTime)<<(int)Limit;//inform the
buffer
                                                    // of the possibility
to
                                              // send
              }
            }
```

```
//*************************************************************************
*****
//                          SEND NEW DATA
//*************************************************************************
*****

            if(input.dataNew)
            {
                //read new data and save it in buffer
              input.get().getMessage(buffer[counter%Win_Size]);
                const TCPPacket* packet=(const TCPPacket *)
                            buffer[counter%Win_Size].myData();
              long num=packet->readNum();
              long seq=packet->readSeq();
if(debug)
OUTPUT<<" SEND "<<seq<<" NUM "<<num<<" at "<<completionTime<<"\n";
              output.put(completionTime)<<buffer[counter%Win_Size];

              if(num>=Limit-MSS) //sliding window full?
              {
                state=STOP;
                demand.put(completionTime)<<(int)state;
              }
              if(Timer==-2) // if no RTT is being measured, start a new
                            // measurement
                {
                  Start_Time=arrivalTime;
                Timer=seq;
              }
                timer_buf[seq%Win_Size]=completionTime+RTO;//set the time
at
                                                    //which a timeout should
occur
              Last_Sent=num;
              counter++;

            }

//*************************************************************************
*****
//                          Timeout for a packet has occured
//*************************************************************************
*****

            if(timer.dataNew)
            {
                Envelope envp;
              timer.get().getMessage(envp);
                const TCPPacket* packet=(const TCPPacket *)
                            envp.myData();
                long seq=packet->readSeq();
              long num=packet->readNum();

                if(num>Last_Ack) // has this packet already been
acknowledged
                {

//*************************************************************************
*****
```

```
//Check if the timeout occured at the right time. If it is not correct
then
// this indicates that the the packet was retransmitted or can not be
//acknowledged now as the receiver is waiting for a retransmission
//**************************************************************************
*****

                    if((completionTime+(0.01)<timer_buf[seq%Win_Size])||
                   (timer_buf[seq%Win_Size]<0))
                         return;
                Retransmitted=last_seq+1;//register the packet that got
lost
if(debug)
OUTPUT<<" TIMEOUT "<<seq<<" Resend "<<Retransmitted<<" NUM "<<num<<" at
"<<completionTime<<"\n";
//**************************************************************************
*****
//clear all the timeout stamps in order to avoid timeout handling for
packets
//that were transmitted but can not be acknowledged until a retransmitted
//packet reaches the receiver
//**************************************************************************
*****
                    long i;
                  for(i=0;i<Win_Size;i++)
                         timer_buf[i]=-1;
                    Retransmission=4;
                  Last_Seq=seq+1;

last_retransmitted=((float)Retransmitted)*(float)MSS;

                  Timer=-2;          //stop the RTO calculation
                  send_state=-1;
                  Limit=0;
                  Slow_Start=1;  //go into slow start
                  Congestion_Avoidance=1; //start the congestion
                                          //avoidance state after
                                       //the slow start
                  if(cwnd<Win)    //reset the ssthresh value
                  {
                        ssthresh=cwnd/2;
                      ssthresh-=(long)ssthresh%MSS;
                  }
                  else
                  {
                        ssthresh=Win/2;
                      ssthresh-=(long)ssthresh%MSS;
                  }
                  if(ssthresh<=MSS) ssthresh=MSS+MSS;// set the lower
                                                     // limit of ssthresh
                   cwnd=MSS;                        // initialize cwnd
                  demand.put(completionTime)<<(int)STOP;
                  Time_Out=1;
                  RTO*=2;      // exponential backoff
                  if(RTO>64)
                     RTO=64;
                  timeout.put(completionTime)<<RTO;// update the time out
                                                   // value

                    timer_buf[seq%Win_Size]=completionTime+RTO;
```

```
                    //resend the last unacknowledged packet

output.put(completionTime)<<buffer[(last_seq+1)%Win_Size];
                }
              }
        } // end go
} // end defstar
```

4. TCP_Dump:

```
defstar {
      name {TCP_Dump }
      domain { DE }
      author { Dorgham Sisalem }
        copyright { SEE $PTOLEMY_SIMULATIONS/TCP/copyright }

      desc { This star reads in a TCP packet and output its contents.
      }

      input { name { input } type { message} }
      output { name { seq} type { int} }
      output { name { ack} type { int} }
      output { name { win} type { int} }

      defstate {
            name { logfile}
            type { string}
            desc { }
      }

      hinclude { "TCPPacket.h" ,<fstream.h>}

      protected { long count;
              ofstream log;}

        setup { delayType= FALSE;
              log.open(logfile);
              }

        wrapup { log.flush(); }

      go {
                completionTime= arrivalTime;
            Envelope Envp;
            input.get().getMessage(Envp);

            const TCPPacket* PacketPtr=(const TCPPacket *)
              Envp.myData();
            seq.put(completionTime)<<PacketPtr->readSeq();//output
sequence
//          ack.put(completionTime)<<PacketPtr->readAck();//output
acknowledgement
//          win.put(completionTime)<<PacketPtr->readWin();//output window
size


      } // end go
} // end defstar
```

5. TCP_Rec:

```
defstar{
      name { TCP_Rec }
      domain { DE }
      author { Dorgahm Sisalem }

      desc {
        This star simulates a TCP receiver that fulfills the following
        requirements:
        Sliding window: Using  acknowledgment packets, the receiver
informs
                the source of the allowed number of bytes that can be
sent
             before the sender buffer gets full.
        Update packets: After sending an acknowledgment with the window
size
                smaller than the maximum segment size (MSS) an update
packet
                is sent when there is new buffer space available.
        silly window: If the window size gets too small -smaller than
MSS-
             an update packet is only sent after freeing enough buffer
space
             - here, it was set to half the original window size.
        Duplicate acks: If a packet is lost the receiver sends for each
new
                received packet an Ack for the last correctly received
packet.
                All out of order packets get buffered until the missing
packet is
             received. Only then can these packets get also
acknowledged.
        Delayed acks: Ack packets are sent after an ack timer goes off or
every
                other packet.

        In this model it is assumed that all packets have the same length
        -MSS-. Based on this the receiver only informs the application
when a
        packet is received. If there is any interest in using variable
packet
        sizes, the actual received packet must be sent to the
application.
      }

//*************************************************************************
//                Define Ports
//*************************************************************************
      input { name { input }
              type { message}
            desc { new data}
      }
      input { name { Ack_Fin}
              type { int}
            desc { ack timer went off}
      }
```

```
        input { name { Size}
                type { int}
              desc { size of available buffer }
      }

      output { name { control}
                type { message}
               desc { send Ack}
        }
        output { name { application}
                type { int}
               desc { send a packet to the application}
      }
      output { name { Ack_Start}
                type { int}
               desc { start the ack timer }
      }

//************************************************************************
//                Define States
//************************************************************************
      defstate {
            name { Window_Size}
            type { int  }
            default { 10 }
                desc {maximum transmission window. }
      }
      defstate {
               name { fileName }
               type { string }
               default {"" }
      }

      defstate {
            name { MSS}
            type { int  }
            default { 1 }
            desc { Maximum Segment Size}
      }
      defstate {
            name { Ack_Time}
            type { int  }
            default { 1 }
            desc { Start value for the acknowledgment timer.}
      }
      defstate {
            name { debug}
            type { int  }
            default { 0 }
      }

//************************************************************************
//                Protected variables
//************************************************************************
      hinclude { "TCPPacket.h" ,<memory.h>,<fstream.h>}
      protected {
               Envelope theEnvp,Envp;
               TCPPacket* last_Packet;
               int Win_Size,Win_update;
               int Last_Rec,Ack_Flag;
```

```
                float last_TS;
                int *buffer,Retransmit;
                  int seq,num,size,buf_size,Last_Num,Last_Win;
                int counter,pointer,VC,ECN;
              }
        hinclude { "pt_fstream.h" }
        protected {
              pt_ofstream *p_out;
        }

//*************************************************************************
******/
// Initialization and buffer allocation
//*************************************************************************
******/

      setup
      {
        Ack_Flag=-1;
        Last_Rec=-1;
        Last_Num=0;
        buf_size=0;
        delayType= FALSE;
        Win_update=0;
        Win_Size=Window_Size;
        Window_Size=(int)Window_Size/(int)MSS;
        Retransmit=0;
        counter=pointer=0;
        ECN=0;

      }
        begin{
          if(buffer) {LOG_DEL; delete [Window_Size] buffer; buffer=NULL;}
        LOG_NEW;
        buffer=new int[Window_Size];
        for(int i=0;i<Window_Size;i++)
                  buffer[i]=0;
          if(p_out){LOG_DEL; delete p_out; p_out=NULL;}
          LOG_NEW; p_out = new pt_ofstream(fileName);
        }

        wrapup{
              if(buffer) {LOG_DEL; delete [Window_Size] buffer;
buffer=NULL;}
            if(p_out){LOG_DEL; delete p_out;
            p_out=NULL;}
         }

      constructor
      {
          Size.before(Ack_Fin);
        Size.before(input);
        Size.triggers(control);
        Ack_Fin.before(input);
        Ack_Fin.triggers(control);
          p_out = NULL;
        buffer=NULL;
          if(buffer) {LOG_DEL; delete [Window_Size] buffer; buffer=NULL;}
          if(p_out){LOG_DEL; delete p_out; p_out=NULL;}
      }
```

```
        destructor { if(p_out){LOG_DEL; delete p_out; p_out=NULL;}
              if(buffer) {LOG_DEL; delete [Window_Size]
buffer;buffer=NULL;}
      }
//*************************************************************************
//                GO
//*************************************************************************

      go{
        completionTime= arrivalTime;
          pt_ofstream& OUTPUT = *p_out;

//*************************************************************************
****
//                    update the window size
//*************************************************************************
****

        if(Size.dataNew)
          {
            int actual_win=int(Size.get());
         // if(!counter)                    // update only if nothing is
            Win_Size=(int)(Window_Size*MSS)-actual_win-
((float)counter*(float)MSS);//saved in the local buffer

          if(Win_update)  //handle silly window and send an update packet
            if(float(actual_win)/float(Window_Size)<=0.5)
            {

                const TCPPacket* Packet=(const TCPPacket *)
                          (theEnvp).myData();
            int seq=Packet->readSeq();
            int num=Packet->readNum();
                last_Packet->setDestination(Packet->readDestination());
                last_Packet->setSource(Packet->readSource());
                last_Packet->setVC(Packet->readVC());
            last_Packet->setSize(Packet->readSize());
                last_Packet->setWin(Win_Size);
              last_Packet->setSeq(Last_Rec);
              last_Packet->setAck(Last_Num);
            last_Packet->setTimestamp(Last_Rec);
            last_Packet->setECN(Packet->readECN());
              Envelope outEnvp( *last_Packet);
            Win_update=0;
            control.put(completionTime)<<outEnvp;
                  }
        }

//*************************************************************************
****
//                      new data received
//*************************************************************************
****

        if(input.dataNew)
        {
            //read the new packet
            input.get().getMessage(theEnvp);
          const TCPPacket* PacketPtr = (const TCPPacket*)
                                    (theEnvp).myData();
```

```
            last_Packet=new TCPPacket(*PacketPtr);
            seq = PacketPtr->readSeq();
            num=PacketPtr->readNum();
            size=PacketPtr->readSize();
            VC=PacketPtr->readVC();
            ECN=PacketPtr->readECN();
if(debug)
OUTPUT<<" SEQ "<<seq<<" NUM "<<num<<" ECN "<<ECN<<" at
"<<completionTime<<"\n";            if(Last_Rec>=seq)
            {
                  last_Packet->setNum(num);
                  last_Packet->setSeq(seq);
                  last_Packet->setVC(VC);
                  last_Packet->setWin(Win_Size);
              last_Packet->setECN(PacketPtr->readECN());
                  Envelope outEnvp( *last_Packet);
                  control.put(completionTime)<<outEnvp;
              return;

            }

              int STOP=0;
            if(Last_Rec+1<seq)              //missing packets ??
            {
              for(int i=0;(i<counter)&&!STOP;i++)
              {
                    if(buffer[i%(int)Window_Size]==(int)seq)
                   return;
              if(i+1<counter)
              {
                  if((buffer[i%(int)Window_Size]<seq)&&
                    (buffer[(i+1)%(int)Window_Size]>seq))
                  {
                    for(int tmp=counter-1;tmp>i;tmp--)
                      buffer[(int)(tmp)%(int)Window_Size]=//reorder
packets
                            buffer[((int)(tmp)+1)%(int)Window_Size];
                    STOP=1;
                          buffer[(int)(i+1)%(int)Window_Size]=seq;
                  }
              }
              else
              {
                          buffer[(int)(counter)%(int)Window_Size]=seq;
              }
              }
              if(!counter)
                 buffer[(int)(counter)%(int)Window_Size]=seq;


              if(counter+1>=Window_Size)
              {
              Error::abortRun(*this,"receiver Buffer overflow","");
              return;
              }
              counter++;
                Win_Size-=size;

                //send a duplicate ack
                Retransmit++;   //count the number of out of sequence
```

```
                        //packets
                last_Packet->setAck(Last_Num);
                last_Packet->setSeq(Last_Rec);
                last_Packet->setVC(VC);
                last_Packet->setWin(Win_Size);
                last_Packet->setTimestamp(Last_Rec);
                last_Packet->setECN(ECN);
                Envelope outEnvp( *last_Packet);
                control.put(completionTime)<<outEnvp;
                  return;
              }
              else  // a correct packet was received
              {
                Win_Size-=size;
                if(Retransmit)
                {
                    //send for each out of sequence packet a notification to
the
                  //application
                  application.put(completionTime)<<seq;
                buf_size=0;
                last_TS=Last_Rec+1;

//*************************************************************************
****/
// Control if the buffered packets are correctly received
//*************************************************************************
****/


while((buffer[(int)pointer%(int)Window_Size]==seq+(++pointer))&&
(counter!=0))
                {

                    buf_size+=MSS;
                    application.put(completionTime)<<
                              buffer[(int)(pointer-1)%(int)Window_Size];
                  counter--;
                      Last_Rec=buffer[(int)(pointer-1)%(int)Window_Size];
                }
                pointer--;
                if(!counter) // no packet loss was detected
                  Retransmit=0;

                if(!pointer) // the first packet in the buffer was already
                  Last_Rec=seq; // out of order, so set the received packet
                                // as the last correctly received packet

                Ack_Flag=Last_Rec;
                Last_Num=num+buf_size;
                last_Packet->setSeq(Last_Rec);
                last_Packet->setAck(Last_Num);
                last_Packet->setWin(Win_Size);
                last_Packet->setECN(ECN);

                int i=0;
                while((buffer[(int)pointer%Window_Size]!=0)&&(pointer!=0))
                                                        //reorder the
packets
                {                               // that were left in the buffer
```

```
                buffer[(int)(i++)%Window_Size]=
                            buffer[(int)pointer%Window_Size];
                buffer[(int)(pointer++)%Window_Size]=0;
              }
              pointer=0;

            }
              else
            {
              if(seq<=Last_Rec) //a packet has been received that has
already
                 Ack_Flag=-2;  // been acknowledged

              else
              {
                   application.put(completionTime)<<seq;
                 Last_Rec=seq;
                 Last_Num=num;
                 Retransmit=counter=buf_size=pointer=0;
              }
            }

              if(Ack_Flag!=-1) // a packet has already been received
              {
              last_TS=seq;
              Last_Win=Win_Size;
                if(Win_Size<=MSS) Win_update=1;   // silly window ?
                if(Win_update)
                {
                    last_Packet->setWin(0);
                  Last_Win=0;
              }
                else
                  last_Packet->setWin(Win_Size);
              last_Packet->setTimestamp(last_TS);
              last_Packet->setVC(VC);
              last_Packet->setECN(ECN);
              Envelope outEnvp( *last_Packet);
              control.put(completionTime)<<outEnvp; // send ack
              Ack_Flag=-1;
              }
              else  //start ack timer
              {
                Ack_Flag=seq;
              last_TS=seq;
              Ack_Start.put(completionTime)<<int(seq);
              }
            }
          }

//*************************************************************************
**
//                              ack timer went off
//*************************************************************************
*

          if(Ack_Fin.dataNew)
          {
             Ack_Fin.dataNew=FALSE;
```

```
            //if the timer went off for a packet that has not been
            //acknowledged yet, send an ack

            if((Ack_Flag==(int)(Ack_Fin%0))&&!Retransmit)
            {
               const TCPPacket* Packet=(const TCPPacket *)
                           (theEnvp).myData();
               int seq=Packet->readSeq();
             int num=Packet->readNum();
             Last_Win=Win_Size;
               if(Win_Size<=MSS) Win_update=1;    // silly window ?
               if(Win_update)
               {
                   last_Packet->setWin(0);
                 Last_Win=0;
             }
               else
                 last_Packet->setWin(Win_Size);
                 last_Packet->setDestination(Packet->readDestination());
                 last_Packet->setSource(Packet->readSource());
                 last_Packet->setVC(Packet->readVC());
                 last_Packet->setSize(Packet->readSize());
             last_Packet->setVC(VC);
               last_Packet->setSeq(seq);

               last_Packet->setAck(num);
             last_Packet->setTimestamp(seq);
             last_Packet->setECN(ECN);
               Envelope outEnvp( *last_Packet);
             control.put(completionTime)<<outEnvp;  // send ack
               Ack_Flag=-1;
            }
           }
      } // end go
} // end defstar
```

6. ReceiverWin:

```
defstar {
      name { ReceiverWin}
      domain { DE }
      author { Dorgham Sisalem }
        copyright { SEE $PTOLEMY_SIMULATIONS/TCP/copyright }

        desc   { This star simulates the buffer of the application.
                 It is assumed that the packets always have a constant
size
                -MSS-. The buffer only receives the sequence number from
the
                TCP receiver. When the application is ready for a new
packet
                  the size of the window is changed bu MSS
                If there is any interest in variable packet sizes
                then the receiver must be modified to send real packets.}

      input { name { demand}
                type { int }
```

```
            desc { the application has finished consuming the last
packet}
      }
      input { name { inData }
                type { anytype}
              desc { new data}
      }

      output {
            name {outData}
            type {=inData}
              desc { send data to the application}
      }
      output {
            name {size}
            type {int}
            desc { Size of Queue. Triggered by demand inputs. }
      }
      output {
            name {overflow}
            type {=inData}
            desc {
              Arrival data that cannot be queued due to capacity limit.
            }
      }

      protected {
            int infinite;
      }
      defstate {
            name {capacity}
            type {int}
            default {"-1"}
            desc { Maximum size of the queue. If <0, capacity is
infinite. }
      }
      defstate {
            name {MSS}
            type {int}
            desc { Maximum segment size}
      }

      constructor {
            // Indicate to the scheduler that size outputs are triggered
            // demand inputs.
            // simultaneous inData events should be available in the
            // same firing with demand events
               demand.before(inData);
      }
      setup {
            infinite = (int(capacity) < 0);
            control=1;
            Q_Size=0;
            zapQueue();
      }

      ccinclude {
            "DataStruct.h","TCPPacket.h"
      }
      protected {
```

```
        Queue queue;
}
method {
        name { enqueue }
        access { protected }
        arglist { "()" }
        type { void }
        code {
            if (infinite || Qsize() < int(capacity)) {
              queue.put(inData.get().clone());
            } else {
              // Data is lost
              overflow.put(completionTime) = inData.get();
            }
        }
}
method {
        name { dequeue }
        access { protected }
        arglist { "()" }
        type { Pointer }
        code {
            return queue.get();
        }
}
method {
        name { handleOverflow }
        access { protected }
        arglist { "()" }
        type { void }
        code {
            // Nothing to do
        }
}
method {
        name { Qsize }
        access { protected }
        arglist { "()" }
        type { int }
        code {
            return queue.length();
        }
}
method {
        name { zapQueue }
        code {
              while (Qsize() > 0) {
                    Particle* pp = (Particle*) queue.get();
                    pp->die();
              }
              queue.initialize();
              }
          }
protected{
  int control;
  int Q_Size;
}
go {
    completionTime = arrivalTime;
```

```
//*************************************************************************
*****
// The application finished consuming a packet an can take another one
//*************************************************************************
*****

          if (demand.dataNew)
          {
            demand.dataNew= FALSE;
                  if(Qsize())
            {
            dequeue();
            Q_Size-=MSS;  //update the buffer size
            outData.put(completionTime)<<Q_Size;
            control=0; // application is busy
            }
            else
              control=1; //application is free
            size.put(completionTime)<<Q_Size; // update the window size
          }

//*************************************************************************
**
// If there is new inData, store it if the application is busy
//*************************************************************************
**

          if (inData.dataNew)
          {
            inData.dataNew=FALSE;
            if(!Qsize()&&control)
            {
            outData.put(completionTime) =inData%0;
              control=0;
            }
            else
            {
            enqueue();
              Q_Size+=MSS;

            }
            size.put(completionTime)<<Q_Size;
          }

      }

        destructor {
            zapQueue();
        }
}
```

ANEXO B
ALGORITMO DE KARN

Si tanto la transmisión original como la mas reciente fallan en proporcionar tiempos de viaje redondo, ¿qué debe hacer el TCP?. La respuesta aceptada es sencilla: el TCP no debe actualizar la estimación de viaje redondo para los segmentos retransmitidos. Esta idea conocida como *Algoritmo de Karn*, evita el problema de todos los acuses de recibo ambiguos únicamente al ajustar la estimación de viaje redondo para acuses de recibo no ambiguos (acuses relacionados con segmentos que solo se transmitieron una vez).

Por supuesto una implantación simplista del algoritmo de Karn, que solamente ignore los tiempos para los segmentos retransmitidos, también puede conducir a fallas. Considere lo que sucede si el TCP envía un segmento después de un aumento significativo en el retraso. El TCP computa una terminación de tiempo mediante la estimación existente de viaje redondo. La terminación de tiempo será demasiado pequeña para el nuevo retardo y forzará la retransmisión. Si el TCP ignora los acuses de recibo para los segmentos retransmitidos, nunca actualizará la estimación y el ciclo continuará.

Para resolver dichas fallas, el algoritmo de Karn necesita que el transmisor combine las terminaciones de tiempo de transmisión con una estrategia de *anulación del temporizador* (timer backoff). La técnica de anulación computa una terminación de tiempo inicial por medio de una formula. Sin embargo si se termina el tiempo y se provoca una retransmisión, el TCP aumenta el valor de terminación de tiempo. De hecho, cada vez que debe retransmitir un segmento, el TCP aumenta el valor de terminación de tiempo.

La mayor parte de implantaciones usan varias técnicas para computar la anulación . la mayor parte escoge un factor multiplicativo, γ, y ajustan el nuevo valor a:

$$new\_timeout = \gamma * timeout$$

Por lo general, γ es 2. (se ha argüido que los valores de γ menores a 2 provocan inestabilidades). Otras implantaciones usan una tabla de factores multiplicativos, lo que permite la anulación arbitraria de cada paso.

El algoritmo de Karn ocupa la técnica de anulación con la estimación de viaje redondo para solucionar el problema de no incrementar las estimaciones de viaje redondo:

Algoritmo de Karn: cuando se compute la estimación de viaje redondo, ignorar los ejemplos que correspondan a los segmentos retransmitidos, pero utilizar una estrategia de anulación, y mantener el valor de terminación de tiempo de un paquete retransmitido para los paquetes subsecuentes, hasta que se tenga un ejemplo válido.

Hablando en forma general, cuando un a red no se comporta adecuadamente, el algoritmo de Karn separa el cómputo del valor de terminación de tiempo de la estimación actual de viaje rdondo. La experiencia demuestra que el algoritmo de Karn funciona bien, inclusive en las redes que tienen alta pérdida de paquetes.

La estimación de viajes redondos ha mostrado que los cómputos descritos con anterioridad no se adaptan a un rango amplio de variación en el retraso. La teoría de poner en cola de espera sugiere que las variaciones en el tiempo de viaje redondo, σ, variarán proporcionalmente a $1/(1-L)$, donde L es la carga actual de la red, $0 \geq L \geq 1$. si una red está corriendo al 50% de su capacidad, esperamos que el retraso de viaje redondo varíe por un factor de $\pm 2\sigma$, o 4. Cuando la carga llega al 80%, esperamos una variación de 10. el estándar TCP original especificaba la técnica para la estimación del tiempo descrito anteriormente y cuyo uso limitando β al valor sugerido de 2 adaptan la estimación de tiempo hasta el 30%.

La estimación de 1989 necesita que las implantaciones estimen tanto el tiempo promedio de viaje redondo como la variación, y que utilicen la variación estimada en vez de constante β. Como resultado, las nuevas implantaciones del TCP se pueden adaptar a un rango mas amplio de variación en el retraso y generar sustancialmente una salida mas alta. Por fortuna las aproximaciones requieren muy poca computación, se pueden derivar programas muy eficientes de las siguientes ecuaciones simples:

$$DIC = SAMPLE - Old\_RTT$$

$$Smoothed\_RTT = Old\_RTT + \delta * DIFF$$

$$DEV = Old\_DEV + \rho(|DIFF| - Old\_DEV)$$

$$Timeout = Smoothed\_RTT + \eta * DEV$$

Donde DEV es la desviacion estimada deseada, δ es una fraccion entre 0 y 1 controlando qué tan rápidamente afecta el nuevo ejemplo al promedio calculado, ρ es una fraccion entre 0 y 1controlando qué tan rápidamente afecta el nuevo ejemplo de desviación estimada deseada, y η es un factor controlando qué tan rápidamente afecta la desviación a la terminación de tiempo del viaje redondo. Para hacer el cómputo mas eficiente, el TCP selecciona δ y ρ para que cada una sea un inverso de una potencia de 2, escala el cómputo por 2^n para lograr n apropiadamente , y utiliza aritmética de números enteros. La investigación sugiere que los vlores de $\delta = 1/2^3$, $\rho = 1/2^2$, y $n = 3$ funcionarán bien.