PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

ESTUDIO DE DESEMPEÑO DEL PROTOCOLO SCTP (STREAM CONTROL TRANSMISSION PROTOCOL) Y ANÁLISIS COMPARATIVO CON LOS PROTOCOLOS TCP Y UDP

TESIS PARA OPTAR EL TÍTULO DE

INGENIERO DE LAS TELECOMUNICACIONES

PRESENTADO POR

Isaac Fernández Baca Peña

Asesor: Ing Genghis Ríos Kruger

LIMA – PERÚ

2007

RESUMEN

El objetivo de este trabajo de Tesis es realizar un estudio de las características y del desempeño del protocolo SCTP (*Stream Control Transmission Protocol*) y contrastarlo con los protocolos clásicos de la capa de transporte. SCTP es un protocolo que trabaja en la capa de transporte, el cual diferencia de TCP que permite crear un solo canal de transmisión, con SCTP se puede crear diversos canales de conexión que trabajan en paralelo, de esta manera la transmisión es más rápida entre cliente y servidor. Todo el trabajo se realiza con aplicaciones existentes que funcionan bajo el entorno de Linux con el objetivo de mostrar los beneficios al usar SCTP en lugar de TCP, además se muestran herramientas útiles para crear aplicaciones que trabajen sobre SCTP ya que este trabajo de Tesis de Pregrado es el trabajo previo para una posterior Tesis de Maestría o de Doctorado.

D E D I C A T O R I A

A César y Augusta, mis padres

A Laura, Manuel, Rebeca y Oscar, mis abuelos

A Yashamiel y Alondra, mis hermanas

A los lectores

A G R A D E C I M I E N T O S

Quiero agradecer a Dios quien ha puesto a en mi camino a muchas personas que han ayudado a que cada vez sea una mejor persona, y ha puesto a distintas personas que de alguna manera me han ayudado en el desarrollo de este trabajo de Tesis.

A mi padre y a mi madre por ser el apoyo incondicional durante toda mi vida.

A mi abuelo Oscar Peña quien forjó los primeros pasos en mi educación.

A la Licenciada Magally García quien confió en mi e hizo posible mi permanencia en esta Universidad.

Al Ingeniero Genghis Ríos Kruger por ser mi asesor y apoyarme en todas las correcciones de la Tesis y en el planteamiento de la misma.

Al Ingeniero Flavio Ramírez, por ser mi asesor, brindarme la logística necesaria y estar apoyándome durante la etapa de pruebas.

Al Ingeniero Arturo Díaz, quien siempre estuvo dispuesto para resolver algunas dudas en Linux.

Al Ph.D Gustavo Do Carmo, quien es un experto en SCTP, y me facilitó bastante información durante todo el desarrollo de la Tesis y me permitió transcribir su código fuente del EchoTools. Muito Obrigado Gustavo!!!

Al Ph.D Elvis Pfützenreuter, experto en SCTP, quien resolvió muchas de mis dudas. Thanks a lot Elvis.!!!

Al Ingeniero David Chávez y al Dr. Carlos Silva, quienes fueron mis tutores de Tesis, y siempre estuvieron dándome consejos para alentarme y terminar el trabajo pronto.

A la preciosa Angelita por siempre darme aliento.

A los lectores, que sin ustedes este trabajo no es más que papel muerto.

Muchísimas Gracias a todos de corazón!!!

I N D I C E

DEDICATORIA I
AGRADECIMIENTOS II
INDICE III
LISTA DE FIGURAS V
LISTA DE TABLAS VII
GLOSARIO VIII

OBJETIVOS 1
INTRODUCCION 2

CAPITULO I PROTOCOLOS DE LA CAPA DE TRANSPORTE 3
1.1 Protocolo UDP 4
1.2 Protocolo TCP 5
1.2.1 Las tres fases en la comunicación TCP 5
1.2.2 Los estados de la conexión en TCP 8
1.3 Protocolo SCTP 11
1.3.1 El formato del paquete SCTP 11
1.3.2 Los estados de SCTP 15
1.3.3 Características de SCTP 18
1.4 Comparación de UDP, TCP y SCTP 27
1.4.1 Limitaciones de UDP 27
1.4.2 Limitaciones de TCP 28
1.4.3 SCTP vs. TCP vs. UDP 28
1.4.3.1 Semejanzas entre TCP y SCTP 30
1.4.3.2 Diferencias entre TCP y SCTP 31

CAPITULO II PROGRAMACIÓN DE SOCKETS EN LENGUAJE C SOBRE LINUX 36

2.1 El *SOCKET* 38
2.1.1 Tipos de *Sockets* 39
- *Sockets* de Flujo *SOCK_STREAM* 39
- *Sockets* de Datagrama *SOCK_DGRAM* 39
- *Sockets* de Paquetes Secuenciales *SOCK_SEQPAQUET* 39

2.1.2 Las estructuras en los Sockets 39
2.1.3 Las conversiones en los *Sockets* 40
2.1.4 Algunas de las funciones utilizadas en la programación de *Sockets* 42
- Función *socket()* 42
- Función *bind()* 43
- Función *connect()* 44
- Función *listen()* 44
- Función *accept()* 45

2.2 Vista breve del modelo Cliente – Servidor 45
2.3 API de sockets para SCTP 46
2.3.1 Sockets SCTP - TCP-style interface 47
2.3.2 Sockets SCTP - UDP-style interface 52
- Función sctp_bindx*()* 53

- Función sctp_connectx*()* 53
- Función sctp_sendmsg*()* 54
- Función sctp_recvmsg*()* 55
- Ejemplos de código fuente usando UDP-*style* 55

CAPITULO III SELECCIÓN E IMPLEMENTACIÓN DE LA APLICACIÓN 57
3.1 Selección de la distribución de Linux y del IDE 58
3.2 Software existente que ya utiliza SCTP 60
3.2.1 Iperf 61
3.2.2 Netperf 62
3.2.3 SCTPperf 62
3.2.4 EchoTools 64

CAPITULO IV ANÁLISIS DE COMPARACIÓN DEL DESEMPEÑO DE SCTP CON TCP Y UDP 66

4.1 Pruebas con Iperf 66
4.2 Pruebas con EchoTools 69
4.2.1 Variando el tamaño de los mensajes 72
4.2.2 Variando el número de los mensajes 75
4.3 Pruebas con SCTPperf 76
4.3.1 SCTPperf sobre ethernet 77
4.3.2 SCTPperf sobre wireless (802.11b) 77
4.3.3 SCTPperf sobre ethernet y *wireless* (802.11b) - *Multihoming* 78
4.3.4 SCTPperf *Multihoming* con varias interfaces de red 81

CONCLUSIONES 84
OBSERVACION 85
RECOMENDACIONES PARA TRABAJOS FUTUROS 86
BIBLIOGRAFÍA 87
ANEXOS 89

L I S T A D E F I G U R A S

Número *Página*

Figura1.1. MODELO TCP/IP [1] 4
Figura1.2. CAMPOS UDP [1] 4
Figura1.3. ESTABLECIMIENTO DE LA CONEXIÓN EN TCP [2] 6
Figura1.5. TÉRMINO DE LA CONEXIÓN EN TCP [2] 8
Figura1.6. DIAGRAMA DE ESTADOS EN TCP[2] 10
Figura1.7. FORMATO DEL PAQUETE SCTP [4] 12
Figura1.8. FORMATO DEL DATA CHUNK [5] 13
Figura1.9. DIAGRAMA DE ESTADOS EN SCTP [5] 16
Figura1.10. ESCENARIO DE TRABAJO DE LOS ESTADOS EN SCTP [5] 17
Figura1.11. CONEXIÓN TCP VS. ASOCIACIÓN SCTP [6] 19
Figura1.12. RELACIÓN DE UNA ASOCIACIÓN SCTP Y LOS STREAMS[6] 20
Figura1.13. ESTABLECIMIENTO DE LA COMUNICACIÓN SCTP[7] 22
Figura1.14. ATAQUE DE INUNDACIÓN CON SEGMENTOS SYN [8] 23
Figura1.15. MESSAGE FRAMING EN UDP/SCTP VS. TCP [6] 24
Figura1.16. SECUENCIA DE MENSAJES PARA EL TÉRMINO DE LA COMUNICACION EN TCP Y SCTP. [6] 25
Figura1.17. GRACEFUL SHUTDOWN[8] 26
Figura1.18. ESTABLECIMIENTO DE LA COMUNICACIÓN EN TCP Y SCTP[6] 30
Figura1.19. UNA CONEXÍON EN TCP[7] 31
Figura1.20. UNA ASOCIACIÓN EN SCTP[7] 32
Figura1.21. EL PROBLEMA “ HEAD-OF-LINE BLOCKING ”[10] 33
Figura2.1. ENVIO DE UN CARTA[9] 36
Figura2.2. DOS SIMPLES PROGRAMAS DE NETWORKING[9] 38
Figura2.3. BIG-ENDIAN BYTE ORDER y LITTLE-ENDIAN BYTE ORDER.[5] 41
Figura2.4. TRANSFORMACIÓN DE BYTE ORDER[5] 42
Figura2.5. OPCIONES DE LOS PARAMETROS DE LA FUNCION SOCKET[2] 43
Figura2.6. DIAGRAMA DE FLUJO CLIENTE-SERVIDOR TCP[2] 48
Figura2.7. DIAGRAMA DE FLUJO CLIENTE-SERVIDOR SCTP[2] 49
Figura2.8. CAPTURA DE TRAMAS CLIENTE/SERVIDOR SOBRE TCP 51
Figura2.9. CAPTURA DE TRAMAS CLIENTE/SERVIDOR SOBRE TCP 51
Figura2.10. DIAGRAMA DE FLUJO CLIENTE-SERVIDOR SCTP- UDP STYLE SOCKET[2] 52
Figura2.11. RESULTADOS CLIENTE-SERVIDOR SCTP- UDP STYLE SOCKET 56
Figura2.12. CAPTURA CLIENTE-SERVIDOR SCTP- UDP STYLE SOCKET 56
Figura2.13. . 57
Figura4.1. ESCENARIOS DE PRUEBAS DE COMPARACIÓN TCP VS SCTP 71
Figura4.2. ESCENARIO DE PRUEBA DE MULTIHOMING 76

Figura4.3. CAPTURA DE TRAMAS EN EL ESCENARIO MULTIHOMING 81
Figura4.4. ESCENARIO MULTIHOMING CON TRES INTERFACES DE RED 82
Figura4.5. CAPTURA ESCENARIO MULTIHOMING CON TRES INTERFACES DE RED 83

L I S T A D E T A B L A S

Número *Página*

Tabla 1.1 DEFINICIÓN Y PARÁMETROS DE LOS *CHUNKS* DE CONTROL.....14

Tabla 1.2 COMPARACION DE LAS CARACTERISTICAS DE SCTP, TCP Y UDP .29

G L O S A R I O

ACK	*Acknowledge.*
AIMD	*Additive Increase Multiplicative Decrease*
API	*Application Programming Interface*
DNS	*Domain Name Server*
DoD	*Department of Defense, EEUU*
DoS	*Denial of Service*
GPL	*GNU General Public Licence*
GNU	*GNU's not Unix*
HOL	*Head-of-line Blocking*
IETF	*Internet Engineering Task Force*
IP	*Internet Protocol*
ISN	*Initial Sequence Numbers*
MSS	*Maximum Segment Size*
MTU	*Maximum Transmission Unit*
OSI	*Open Systems Interconnection*
PSNT	*Public Switched Telephone Network*
QoS	*Quality of Service*
RAM	*Random-access Memory*
RFC	*Request for Comments*
SACK	*Selective Acknowledge*
SCTP	*Stream Control Transmission Protocol*
SYN	*Sincronización.*
TCP	*Transmission Control Protocol*
TSN	*Transmission Sequence Number*
TTL	*Time to Live*
UDP	*User Datagram Protocol*

O B J E T I V O S

- Estudiar los protocolos UDP, TCP y SCTP.
- Estudiar las API de programación en sockets para aplicaciones que trabajen con UDP, TCP y SCTP.
- Hacer un análisis de las herramientas necesarias para el desarrollo y el desenvolvimiento del protocolo SCTP.
- Seleccionar las aplicaciones que mejor muestran las características del protocolo SCTP y que también trabajan con TCP.
- Analizar el desempeño de SCTP en comparación con TCP y UDP.

I N T R O D U C C I O N

El constante desarrollo de las Telecomunicaciones exige cambios y mejoras en las prestaciones de los servicios y aplicaciones que la sociedad demanda.

Partiendo de esto, se requiere buscar soluciones más adecuadas y eficientes para el desarrollo de aplicaciones sobre Internet. Este estudio tiene como fin hacer un análisis cuantitativo y cualitativo de una nueva forma de transmisión en Internet a partir del cual se podrá encontrar información valiosa para la implementación de esta nueva tecnología en las aplicaciones existentes y venideras.

C A P I T U L O I
P R O T O C O L O S D E L A C A P A D E T R A N S P O R T E

La capa de transporte esta situada entre la capa de aplicación y la capa de Internet en el modelo TCP/IP, el cual es el estándar en el que se basa Internet, este modelo fue creado por el Departamento de Defensa de EE.UU. (DoD) con la idea de un mundo lleno de cables, microondas, fibras ópticas y enlaces satelitales y dada la necesidad de transmitir data independientemente del estado de los diferentes nodos de la red, el DoD buscaba una transmisión confiable hacia cualquier destino.[1]

La capa de transporte se encarga de ofrecer una comunicación lógica entre procesos de aplicación desde un *host* origen a un *host* destino, los protocolos de esta capa son los encargados de segmentar y reensamblar los datos mandados por la capa superior en el mismo flujo de datos o conexión lógica.[1]

Los protocolos más utilizados de la capa de transporte son UDP (*User Datagram Protocol*) y TCP (*Transmission Control Protocol*), además de estos clásicos protocolos existe el SCTP (*Stream Control Transmisión Protocol*) que se detallarán a continuación.

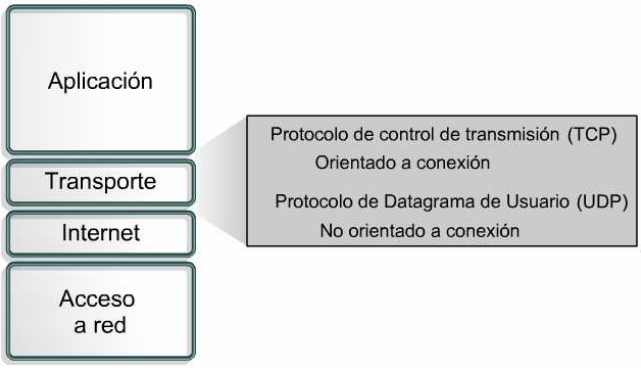


Figura1.1. MODELO TCP/IP [1]

1.1 Protocolo UDP

El protocolo UDP (*User Datagram Protocol*) esta definido para hacer posible el intercambio de datagramas a través de la red. UDP asume que se usa el protocolo IP.

UDP provee procedimiento para que las aplicaciones puedan comunicarse entre ellas con el mínimo mecanismo de protocolo, por lo cual se dice que es un protocolo orientado a la transacción, o contrastándolo con el protocolo TCP, como protocolo no orientado a la conexión, ya que no se garantiza los errores de duplicados, ni la entrega de los paquetes. UDP no usa ventanas ni acuses de recibo, sin embargo la confiabilidad puede ser suministrada por protocolos de la capa de aplicación.

Entre los protocolos que usan UDP tenemos TFTP (Protocolo trivial de transferencia de archivos), (SNMP) Protocolo simple de administración de red, DHCP (Protocolo de configuración dinámica del *host*) y DNS (Sistema de denominación de dominios).[1]

Los campos de un segmento UDP:

Bit 0	Bit 15 Bit 16 Bit 31
Puerto origen (16)	Puerto destino (16)
Longitud (16)	Checksum (16)
Datos (de haber alguno)	

8 bytes

Figura1.2. CAMPOS UDP [1]

- Puerto origen: Número del puerto usado en el *host* origen.
- Puerto destino: Número del puerto usado en el *host* destino.
- Longitud: Número en bytes que consta el encabezado y los datos.
- *Checksum*: Sumatoria calculada en función de los campos del encabezado y de los datos, es utilizada para detectar errores.
- Datos: Datos transportados.

1.2 Protocolo TCP

El Protocolo para el Control de la Transmisión (*Transmission Control Protocol*), es un protocolo de Capa 4, orientado a la conexión, que suministra una transmisión de datos *full-duplex, y es* confiable. TCP forma parte de la pila del protocolo TCP/IP. Se establece una conexión entre ambos extremos antes de iniciar la transferencia de información. TCP brinda un circuito virtual entre las aplicaciones del usuario final. TCP es el responsable de dividir los mensajes en segmentos, reensamblar en el receptor, reenviar cualquier mensaje que no tenga un acuse de recibo, y reensamblar los mensajes a partir de los segmentos recibidos.

A diferencia de UDP que envía los paquetes ni bien empieza, el protocolo TCP tiene tres fases para realizar el envío, ya que la conexión debe ser previamente establecida.

1.2.1 Las tres fases en la comunicación TCP

- **Establecimiento de la conexión**, la cual se debe realizar antes de que comience la transferencia de datos, en esta etapa ambos extremos deben sincronizar sus números de secuencia iniciales (ISN: *Initial Sequence Numbers*), esto se realiza a través del intercambio de segmentos que establecen la conexión al transportar el *bit* SYN (*bit* de control para la sincronización). En la sincronización se intercambian los SYN y un acuse de recibo (ACK), así como se muestra en la figura 1.3.[2]

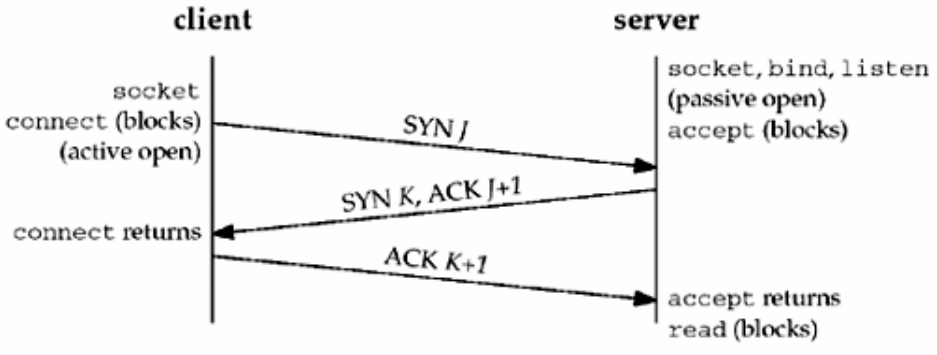


Figura1.3. ESTABLECIMIENTO DE LA CONEXIÓN EN TCP [2]

El establecimiento de la conexión TCP en programación se hace mediante funciones llamadas sockets, lo cual es similar a un sistema de telefonía, en el cual la función *bind* dice a las otras personas cual es nuestro numero de teléfono, entonces ahora ellos pueden llamarnos, la función *listen* es como encender el timbrado, ahora podremos escuchar cuando nos están llamando, la función *connect* es requerida para cuándo la otra persona marca nuestro numero y se intenta comunicar con nosotros, la función *accept* es cuando la persona llamada contesta el teléfono, entonces la conexión esta establecida. La función *connect* se ejecuta el lado del cliente, mientras que las funciones *bind, listen* y *accept* se ejecutan del lado del servidor. [2]

- **Transferencia de datos** en la cual se realiza la transferencia misma de los datos, en esta etapa también se realiza el ordenamiento de los datos transferidos, liberación de los errores, la retransmisión de los paquetes perdidos, el descarte de paquetes duplicados, manejo de la congestión, y control del flujo.[2]

  El ordenamiento de la data se hace usando los ISN ya que este parámetro se incrementa por cada byte transmitido, el ISN es reconocido durante el establecimiento de la conexión, además el ISN es también usado para realizar el descarte de paquetes duplicados y la retransmisión de paquetes perdidos, y haciendo uso de los SACK (*Selective Acknowledge*). Para la liberación de errores, el protocolo TCP utiliza el algoritmo CRC (*Cyclic Redundancy Check*), el cual hace una verificación del campo checksum de

16 bits. El manejo de la congestión y el control de flujo se hacen mediante el uso de ventanas. Una ventana es la cantidad de bytes máximo que puede ser recibido, TCP cuando detecta una congestión reduce la ventana para enviar menos paquetes. De acuerdo al MSS (*Maximun Segment Size*) y el campo *window* se define se puede identificar cuantos paquetes se pueden enviar antes de recibir la confirmación.[1][2][3]

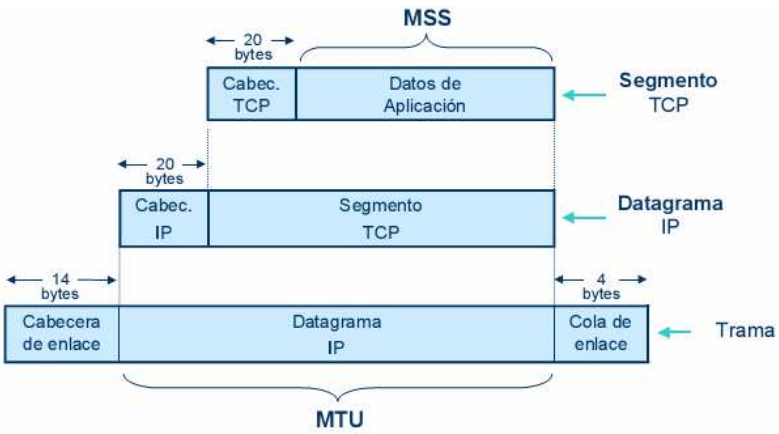


Figura1.4.MAXIMUN SEGEMENT SIZE EN TCP [3]

- **Término de la conexión**, a diferencia del establecimiento de la conexión donde se usaban tres segmentos, se usan cuatro segmentos para el término de la conexión. Tanto el cliente como el servidor pueden terminar la conexión, si consideramos que el cliente es el que envía un *flag* FIN, el servidor le responde con un acuse de recibo, con lo que se detiene el flujo del cliente al servidor, el servidor envía un FIN hacia el cliente, y este responde con otro acuse de recibo, con lo que se detiene el flujo de datos del servidor al cliente. Esto se verá reflejado al llamar la función *close* en la función *main.*[2]

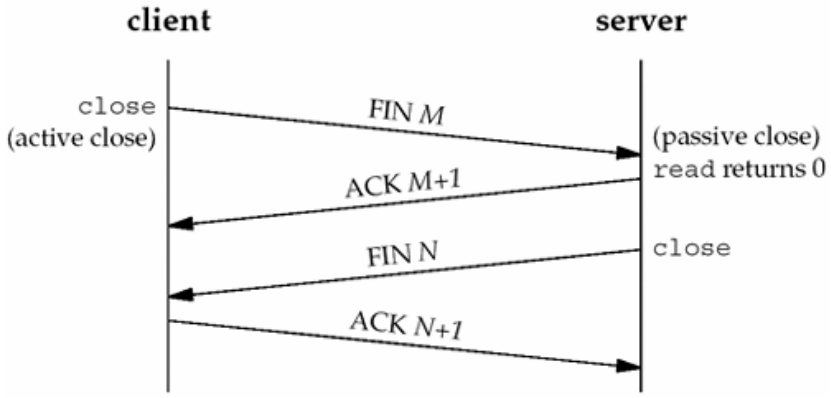


Figura1.5. TÉRMINO DE LA CONEXIÓN EN TCP [2]

1.2.2 Los estados de la conexión en TCP

- CLOSED, Este estado no existe, es usado solo de referencia.
- LISTEN, Espera solicitud de conexión de un TCP remoto.
- SYN-SEND, Esperando un mensaje de solicitud de conexión después de haber enviado una solicitud de conexión.
- SYN-RECEIVED, Esperando una confirmación de un reconocimiento de solicitud de conexión, después de haber enviado y recibido una solicitud de conexión.
- ESTABLISHED, Representa una conexión activa, los datos recibidos pueden ser enviados a un protocolo de capa superior, este es el estado en el que se encuentra en la fase de transferencia de datos.
- FIN-WAIT-1, En es este estado se espera una solicitud de término de conexión TCP, o reconocimiento de que se recibió la solicitud de término de conexión.
- FIN-WAIT-2, En es este estado se espera una solicitud de término de conexión de un TCP remoto.

- CLOSE-WAIT, En es este estado se espera una solicitud de término de conexión de un protocolo de capa superior.

- CLOSING, En es este estado se espera el conocimiento de una solicitud de término de conexión de un TCP remoto.

- LAST-ACK, En es este estado se espera el conocimiento de la confirmación de la solicitud de término de conexión enviada anteriormente al TCP remoto.

- TIME-WAIT, En es este estado se espera el tiempo necesario para que el TCP remoto haya recibido la confirmación de la solicitud del término de conexión.[3]

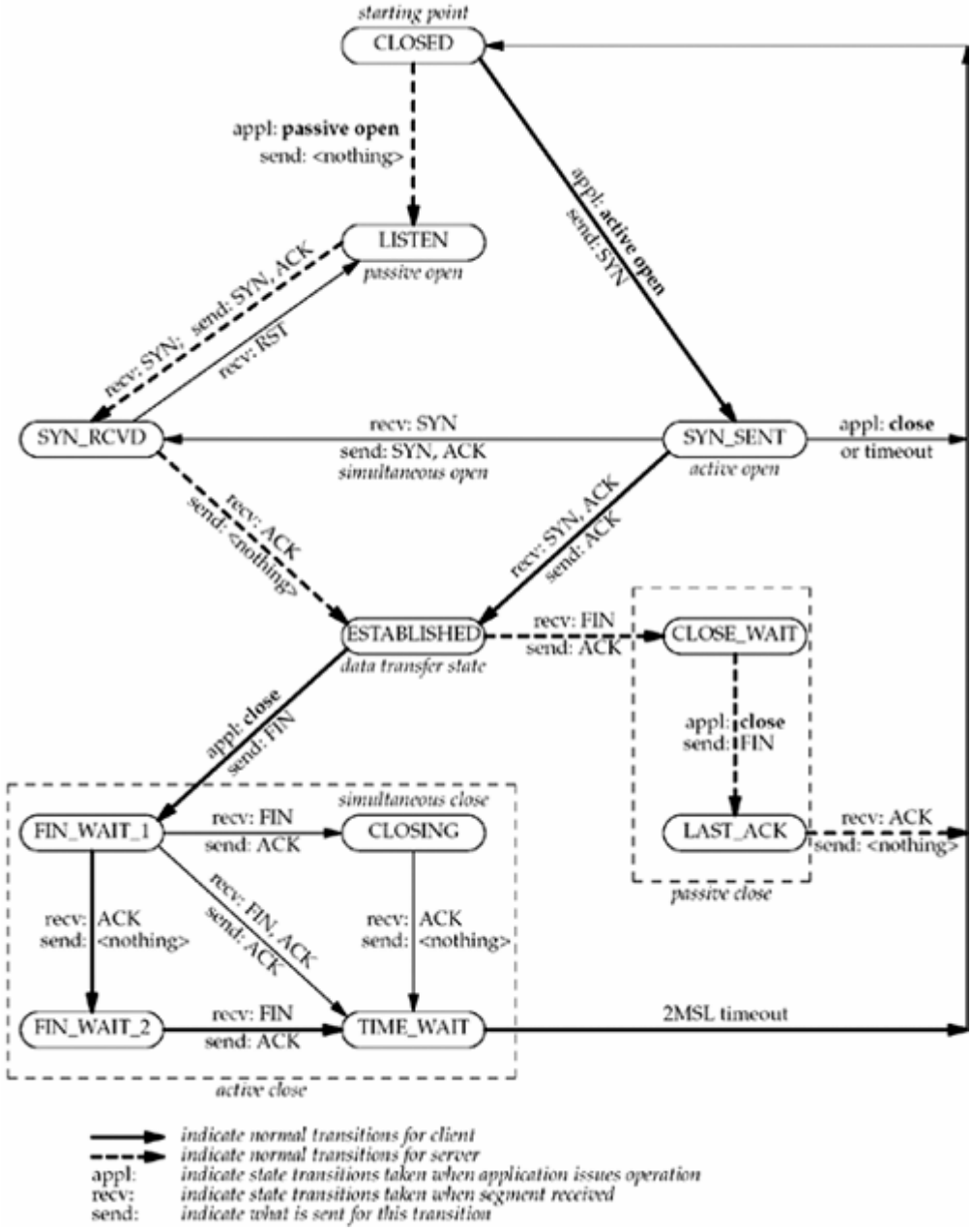


Figura1.6. DIAGRAMA DE ESTADOS EN TCP[2]

1.3 Protocolo SCTP

El protocolo SCTP (*Stream Control Tranmission Protocol*), es un nuevo protocolo para el control de la transmisión, es una adición a la pila de protocolos en la capa 4 del modelo TCP/IP, fue definido por el grupo SIGTRAN de IETF en el año 2000, está descrito en el RFC2960 (*Stewart et al. 2000*). SCTP es un protocolo similar a los protocolos clásicos UDP y TCP, pero que junta las características de ambos para formar un nuevo y mejor protocolo, además que se le agregan más funcionalidades como es el *multi-homing* y el *multi-streaming*. A diferencia de TCP el cual era orientado a la conexión, SCTP es orientado a los mensajes. SCTP provee confiabilidad, control del flujo y secuencia así como TCP, adicionalmente permite el envío de mensajes ordenado o desordenado, a diferencia de TCP que permite solo un envío ordenado y a diferencia de UDP que permite el envío desordenado. El flujo de mensajes de SCTP es similar al envío de paquetes de UDP.[4][14][18]

1.3.1 El formato del paquete SCTP

El paquete SCTP consiste de cabecera SCTP común seguida de una pila de bloques llamados *chunks* (Ver figura 1.7), el número de *chunks* estará limitado por el MTU (*Maximun Transfer Unit*), estos *chunks* puede contener datos de control o la data en sí que se desea enviar.[14]

La cabecera común consta de:

- La dirección del puerto de origen
- La dirección del puerto de destino
- El *tag* de verificación
- El campo *checksum* del paquete

El puerto de origen es usado por el receptor para identificar la asociación a la que pertenece el paquete SCTP, el puerto de destino es el puerto a donde el paquete esta destinado. Ambos extremos asignan un valor de identificación de 32 bits (*verification tag*), este campo es negociado al inicio de la asociación. El *checksum* es una herramienta utilizada para verificar la integridad de cada paquete SCTP, el cual usa el algoritmo *Alder-32,* el cual realiza una suma usando 32 bits, de esta manera se pueden detectar errores.[2][4][14]

Todo tipo de *chunk* tiene los campos: (Ver figura 1.7 – *chunk* 1) :

- **Type**, identifica el tipo de *chunk* que se esta transmitiendo.
- **Flag**, especifica que bits se van a usar en la asociación, depende del tipo de *chunk* que se va ha enviar.
- **Length**, determina el tamaño total del *chunk* en bytes.
- **Chunk data,** incluye la carga de datos que esta llevando el *chunk.*

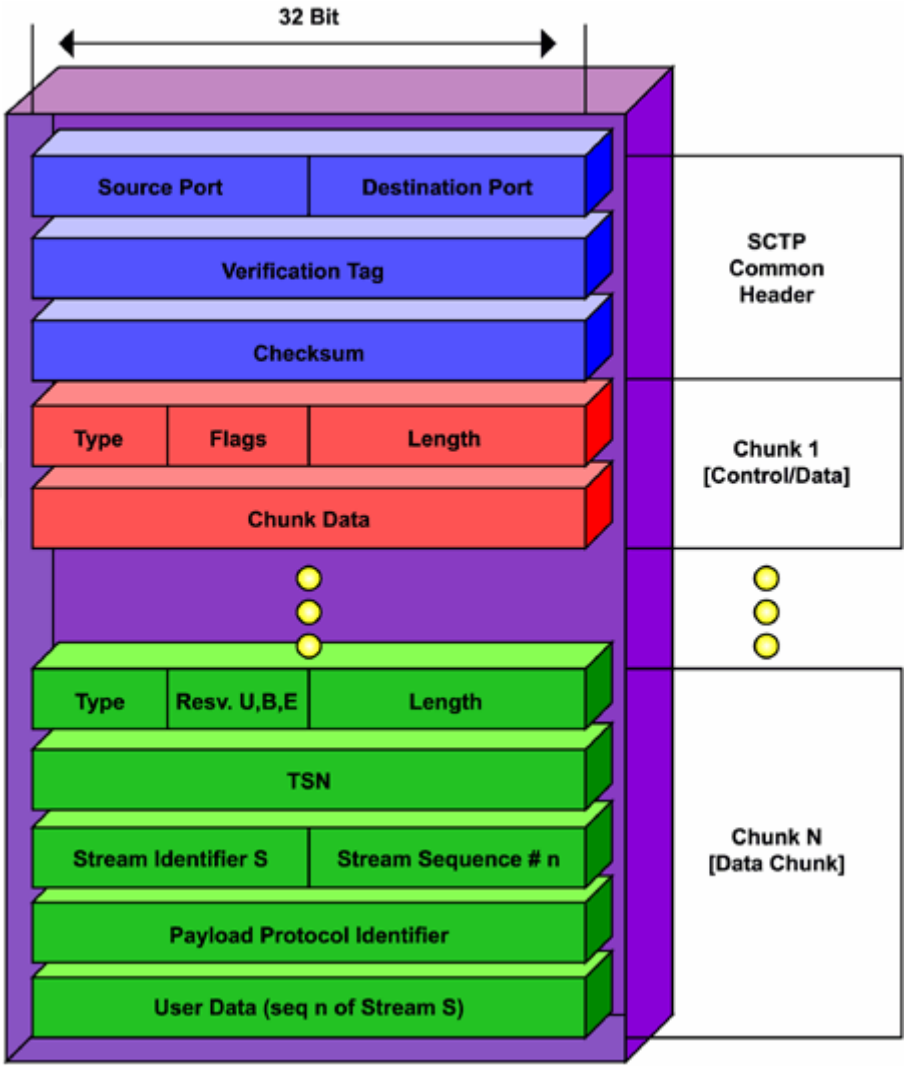


Figura1.7. FORMATO DEL PAQUETE SCTP [4]

Como se puede apreciar en la figura 1.7 un paquete puede tener N *chunks* . SCTP permite que los *chunks* sea multiplexados utilizado la total capacidad del MTU, con la excepción de que necesita los mensajes INIT y INIT-ACK. Son quince tipos de

chunks, definidos en la RFC2960, incluyendo el *data chunk* y otros catorce *chunks* de control. De acuerdo al formato del paquete se puede tener hasta 256 *chunks*, 241 serán definidos en el futuro por el IETF. La definición y parámetros de los *chunks* de control esta resumido en la tabla 1.1., dos de estos *chunks* están reservados para el control de cogestión.[14]

El *DATA CHUNK* es el que lleva los datos que se desean transmitir en sí, el cual debe tener un formato específico (Ver figura 1.8 – *chunk* N), donde el tipo debe tener el valor "0x00", seguido de 5 bits que son reservados, los tres siguientes U, B y E indican respectivamente si la data se va a enviar desordenada, si es el primer *chunk,* y si es el último *chunk.*[14]

El campo TSN (*Transmission Sequence Number*) es utilizado por el transmisor y el receptor para identificar dos *chunks* enviados y recibidos, como auxiliar para retransmisiones y para detectar duplicados, así como también para la fragmentación y re-montaje de los mensajes, el campo *Stream Identifier* identifica el flujo al que pertenece el *chunk*, el campo *Stream Sequence Number* esta asociado al *Stream Identifier,* e identifica el orden de entrega del mensaje, el campo *Payload Protocol Identifier* es ignorado por SCTP, sin embargo, puede ser usado por un usuario para identificar el tipo de información que se esta enviando, e incluso puede ser usado para monitoreo de redes.[14]

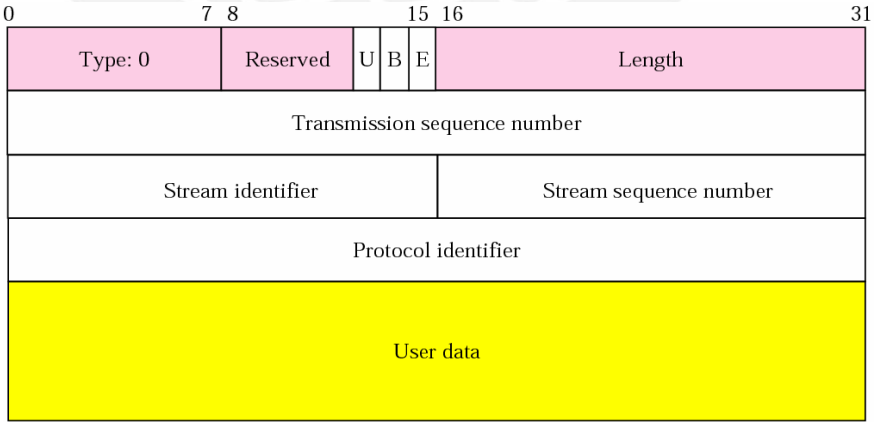


Figura1.8. ***FORMATO DEL DATA CHUNK [5]***

Tabla 1.1 DEFINICIÓN Y PARÁMETROS DE LOS *CHUNKS* DE CONTROL

CHUNK	**DEFINICION**
Initiation INIT	El *chunk* INIT es enviado para inicializar una asociación entre dos terminales.
Initiation Acknowledgement INIT ACK	El *chunk* INIT ACK es un acuse de recibo del *chunk* INIT. El recibo de un INIT ACK establece una asociación.
Selective Acknowledgement SACK	El *chunk* SACK es un acuse de recibo de los DATA *chunks*.
Cookie Echo COOKIE ECHO	El *chunk* COOKIE ECHO es usado exclusivamente durante el proceso de inicio de la asociación.
Cookie Acknowledgement COOKIE ACK	El *chunk* COOKIE ACK es un acuse de recibo del *chunk* COOKIE ECHO. El *chunk* COOKIE ACK debe preceder sobre cualquier *chunk* DATA o *chunk* SACK enviado en la asociación. El *chunk* COOKIE ACK puede ser confirmado con un *chunk* DATA o un *chunk* SACK.
Heartbeat Request HEARTBEAT	Los *chunks* HEARTBEAT son enviados desde un terminal hacia otro con el objetivo de verificar la conectividad hacia específica dirección destino en una determinada asociación.
Heartbeat Acknowledgement HEARTBEAT ACK	Todo el tiempo un *chunk* HEARTBEAT es recibido por un terminal, y responde con un *chunk* HEARTBEAT ACK como acuse de recibo del *chunk* HEARTBEAT.
Abort Association ABORT	El *chunk* ABORT es una indicación de un terminal para forzar terminar la asociación. Adicionalmente un *chunk ABORT* puede informar al receptor las razones o parámetros que causaron la cancelación de la asociación.
Operation Error ERROR	El *chunk* ERROR es enviado por uno de los terminales para reporte un error. El *chunk* ERROR puede contener los parámetros que ayuden a determinar el tipo de error que ocurrió.
Shutdown Association SHUTDOWN	El *chunk* SHUTDOWN es enviado para el terminar la asociación.
Shutdown Acknowledgement SHUTDOWN ACK	El *chunk* SHUTDOWN ACK es el acuse de recibo de *chunk* SHUTDOWN al terminar la asociación.
Shutdown Complete SHUTDOWN COMPLETE	El *chunk* SHUTDOWN COMPLETE concluye el término de la asociación.

(Fuente[4]: The International Engineering Consortium, Stream Control Transmission Protocol)

1.3.2 Los estados de SCTP

- CLOSED, Este estado no existe o no hay conexión, es usado solo de referencia.

- COOKIE-WAIT, En estado se espera una *cookie.*

- COOKIE-ECHOED, En este estado se espera por acuse de recibo del la *cookie.*

- ESTABLISHED, Representa una conexión activa, los datos recibidos pueden ser enviados a un protocolo de capa superior, este es el estado en el que se encuentra en la fase de transferencia de datos.

- SHUTDOWN-PENDING, Enviando la data después de recibir un *close.*

- SHUTDOWN-SENT, Esperando por un acuse de recibo del *shutdown.*

- SHUTDOWN-RECEIVED, Enviando los datos después de recibir el *shutdown.*

- SHUTDOWN-ACK-SENT, Esperando por la completa terminación.[5]

En la figura 1.9 a continuación, se muestra un diagrama de los estados en SCTP.

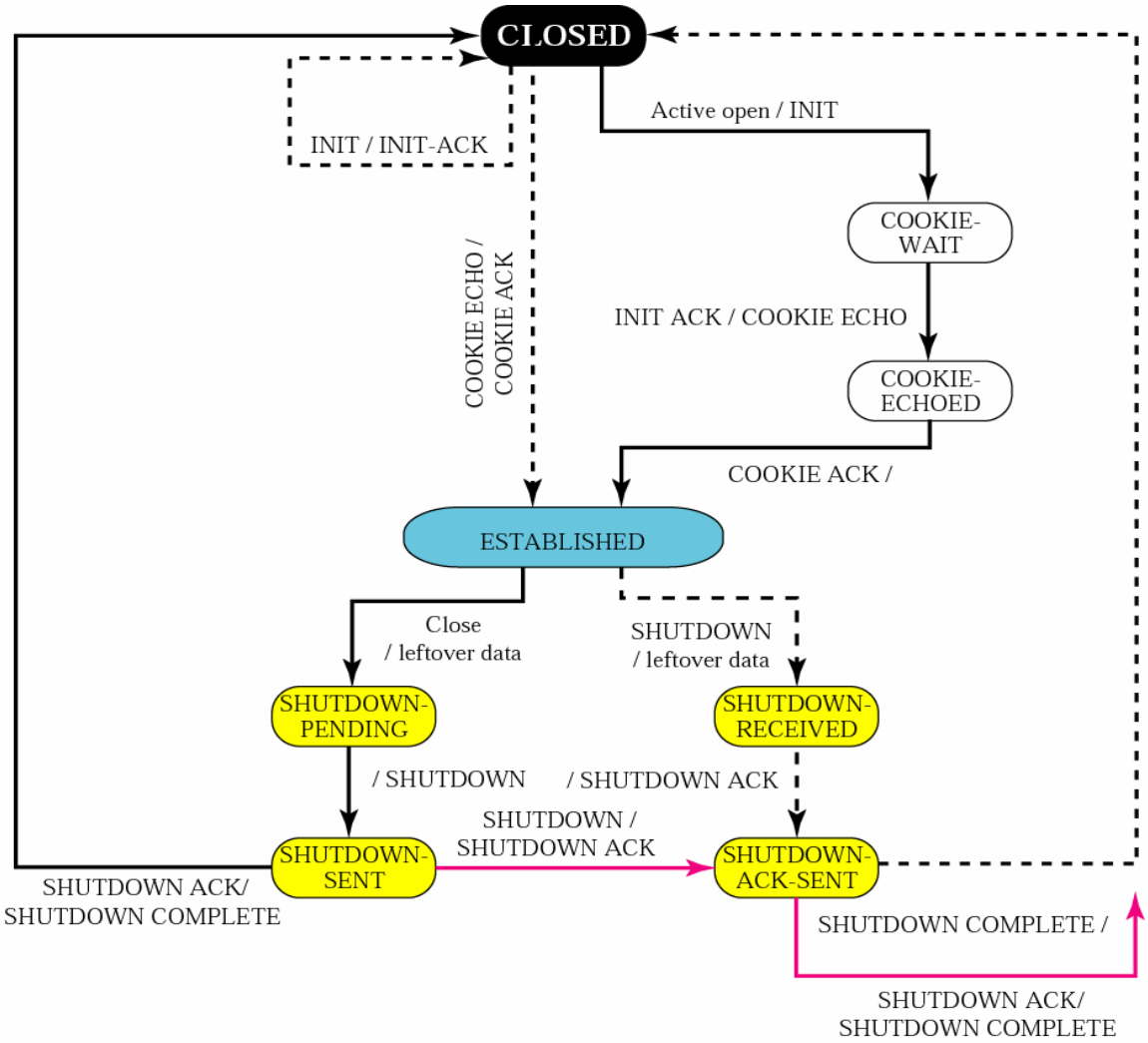


Figura1.9. *DIAGRAMA DE ESTADOS EN SCTP [5]*

En la figura 1.10 se muestra un escenario común de cómo se transita a través de los estado en SCTP, notar que el servidor esta en el estado *CLOSED* hasta que recibe el COOKIE ECHO *chunk,* y que para el término de la asociación hay estados en los que se puede enviar los datos restantes.

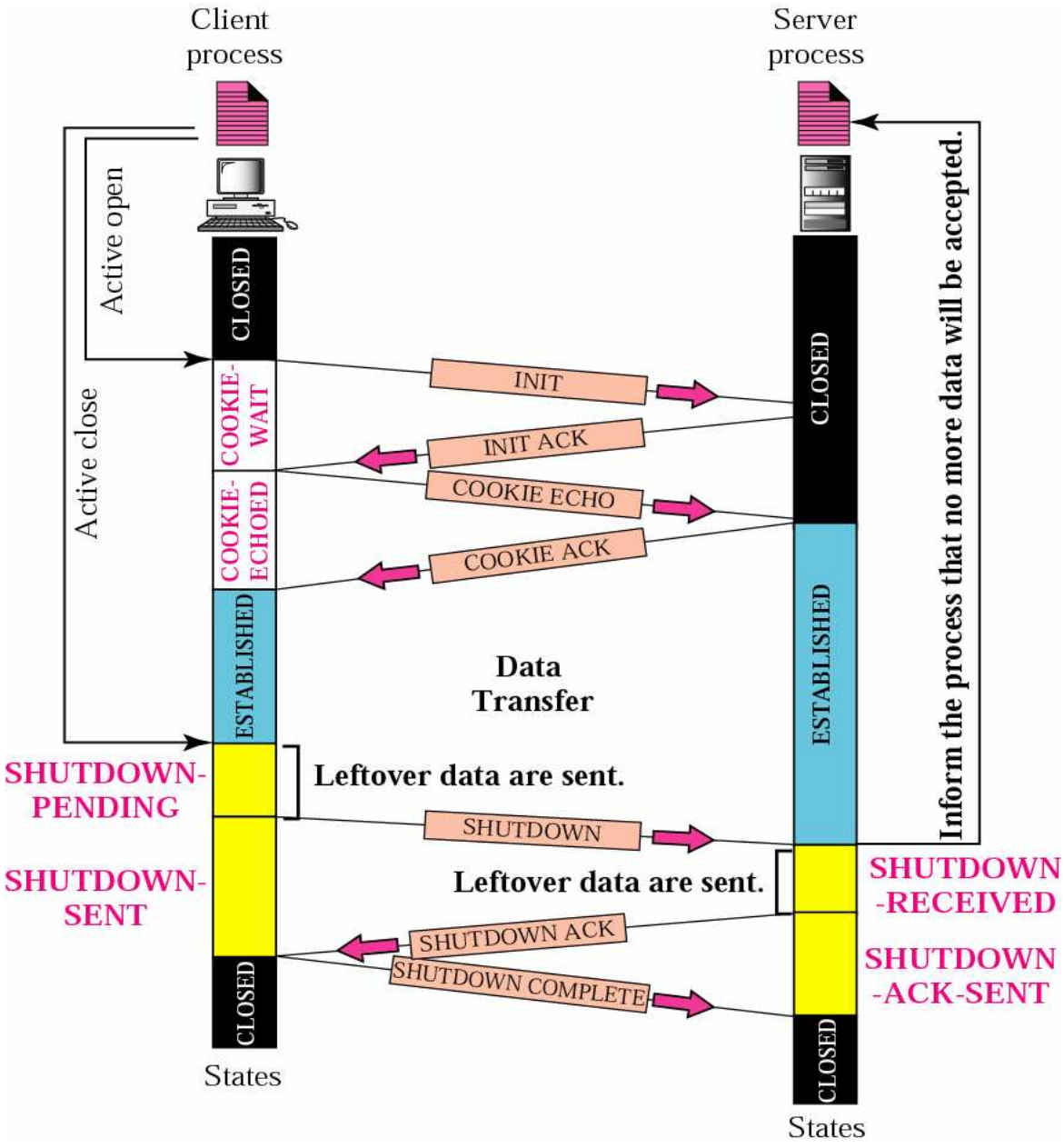


Figura1.10. ESCENARIO DE TRABAJO DE LOS ESTADOS EN SCTP [5]

1.3.3 Características de SCTP

Los dos protocolos más populares son el TCP y UDP, el protocolo TCP es confiable y garantiza el envío y la recepción ordenada de los segmentos y maneja la congestión de la red, UDP es un protocolo no orientado a la conexión o también se puede decir orientado a la transmisión por lo que no garantiza el envío y la recepción ordenada ni la congestión de la red, sino que solo importa transmitir los mensajes, sin importar si todos los paquetes llegaron o no. Sin embargo, UDP es un protocolo rápido y preserva los límites de los mensajes que transporta, es decir envía los mensajes tal cual los recibe de la capa de aplicación, a diferencia de TCP que reacomoda los paquetes de acuerdo al tamaño de ventana de transmisión.[14]

SCTP provee confiabilidad, envío ordenado de los mensajes así como TCP, pero opera de manera similar a UDP, preservando los límites de los mensajes, y provee adicionalmente los siguientes servicios:

- *Multi-homing*
- *Multi-streaming*
- *Initiation protection*
- *Message framing*
- *Configurable unordered delivery*
- *Graceful shutdown*

- **Multi-homing,** permite que las aplicaciones tenga una mayor transferencia de mensajes a que si se usara TCP. Un *multi-homed host* es aquel que tiene más de una interfaz de red, por lo que tiene más de una dirección IP por la cual se puede acceder. En una conexión TCP se refiere a un canal de comunicación entre dos *host*, es un *socket* entre dos *host*. SCTP introduce el concepto de asociación, el cual es un canal de comunicación entre dos *host*, pero permite la colaboración de múltiples interfaces en cada *host.* Veamos la figura 1.11 que ilustra la diferencia entre una conexión y una asociación.[6][14]

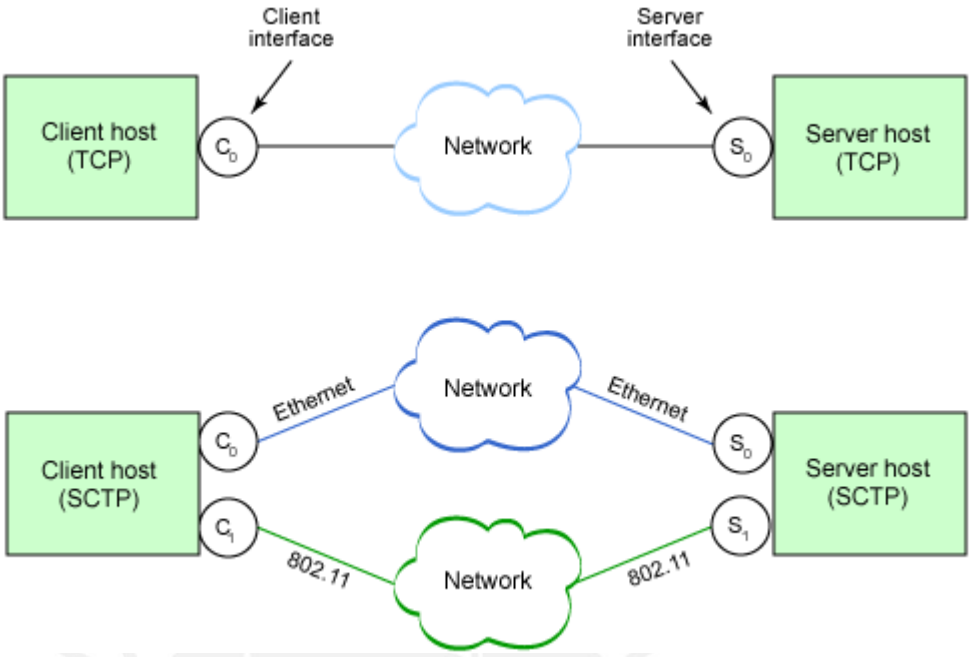


Figura1.11. CONEXIÓN TCP VS. ASOCIACIÓN SCTP [6]

En la parte superior de la figura se muestra una conexión TCP, donde cada *host* cuenta con una única interfaz de red, y se crea una conexión entre el cliente y el servidor. En la parte inferior de la figura se muestra una arquitectura que incluye dos interfaces de red por cada *host*, dos caminos (*paths*) para realizar la comunicación ente el cliente y el servidor. En SCTP estos dos caminos pueden ser usados en una asociación.[6][14]

El protocolo SCTP utiliza ambos caminos para enviar los mensajes, si uno de estos falla, entonces se le considera fuera de servicio, entonces se envía los mensajes por un solo camino – esto es transparente para la aplicación. El protocolo SCTP monitoriza el camino que está fuera de servicio, usando un *heartbeat,* este mensaje se envía periódicamente. Si se recibe un *heartbeat acknowledge* entonces significa que el camino nuevamente esta disponible. SCTP selecciona uno de los caminos como principal *primary address* (el de mayor velocidad). Como todo esto es transparente para la capa de aplicación, podemos dar un ejemplo con el uso de esta característica; si tenemos una computadora portátil que tiene una interfaz inalámbrica 802.11 y una interfaz ethernet, la interfaz ethernet sería la *primary address* por ser la de mayor velocidad, y supongamos que deseamos movernos a otro lugar y

desconectamos la interfaz ethernet por lo que con un TCP y UDP se perdería la conexión, pero con SCTP la comunicación continua por la interfaz inalámbrica, hasta que se vuelva a conectar la interfaz ethernet, entonces detecta esta interfaz y la comunicación nuevamente se da por esta interfaz. Este es un poderoso mecanismo que provee mayor disponibilidad, y que incrementa la confiabilidad. SCTP también permite configurar de modo que se usen todas las interfaces en conjunto.[2][14]

- **Multi-*streaming*,** en forma general una asociación SCTP es como una conexión TCP con lo excepción de que SCTP soporta múltiples *streams* en una misma asociación. Todos los *streams* de la asociación son independientes, pero relacionados en la misma asociación. Veamos la figura 1.12.[6][14]

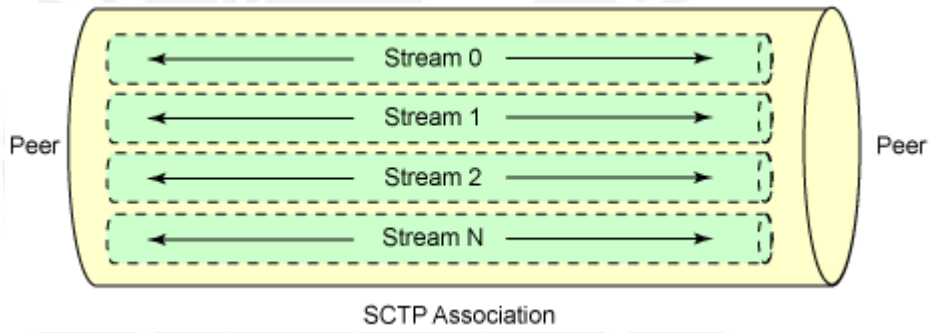


Figura1.12. RELACIÓN DE UNA ASOCIACIÓN SCTP Y LOS STREAMS[6]

Cada *stream* posee su propio número que lo identifica en la codificación en el paquete SCTP que atraviesa la asociación – *stream identifier*. La capacidad de *Multi-streaming* es muy importante, pues la pérdida de un *stream* no afecta a los demás *streams* en la asociación (por ejemplo, cuando se produce un retraso por la espera de la retransmisión de un paquete perdido). Este problema es más conocido como *head-of-line blocking* de TCP, el cual será detallado más adelante en *Comparación de SCTP con UDP y TCP.*[6][7][10][14]

Multi-streaming es una importante funcionalidad de SCTP, especialmente cuando uno considera las características de control del flujo de datos. En

TCP este control y los datos típicamente comparten el mismo canal de conexión, lo cual puede ser problemático porque el control de paquetes puede sufrir un retraso cuando se encuentre tras los datos. En cambio cuando el control y los datos se transmiten en diferentes *streams*, el control se podrá entregar de manera oportuna, dando como resultado una mejor utilización de los recursos.[14]

Un ejemplo que puede mostrar las bondades del *multi-streaming* es un navegador de Internet sobre SCTP. Cuando el protocolo HTTP trabaja sobre TCP utiliza un mismo socket y una sola conexión para el envío de los datos de control y los datos en si. Entonces cuando un cliente Web hace un pedido de una página Web, el servidor le envía todos estos datos por un solo medio, si la pagina tuviera muchas imágenes y archivos, todos estos son enviados por el mismo canal en el que van los datos del control de los mismos. En cambio el navegador de Internet sobre SCTP proveería una mejor interacción ya que un *stream* sería usado para el control, cada archivo e imagen tendría su propio *stream,* de manera que simultáneamente se podría descargar las imágenes y el código HTML, por lo que se observaría un mejor *performance*. Por el otro extremo, un servidor HTTP *multi-streamed*, permitiría diversas conexiones con un mismo *socket,* ofreciendo así un mejor servicio con mayor calidad y rapidez para el acceso a las páginas Web.[14]

- ***Initiation protection*****,** protección al inicio, para empezar a establecer una conexión en TCP – como antes ya se mencionó – se utilizan tres mensajes SYN, SYN-ACK y ACK conocido como *three-way handshake.* En SCTP en cambio son cuatro mensajes INIT, INI-ACK, COOKIE-ECHO, COOKIE-ACK, conocido como *four-way handshake*, veamos la figura 1.13.

  INIT, el cliente envía un mensaje para la iniciación de la asociación

  INIT-ACK, el servidor responde al mensaje INIT con un acuse de recibo y añade una *cookie.*

  COOKIE-ECHO, el cliente responde con la misma *cookie.*

COOKIE-ACK, el servidor reserva recursos para la asociación y responde con un *acknowledge*[2][7][14]

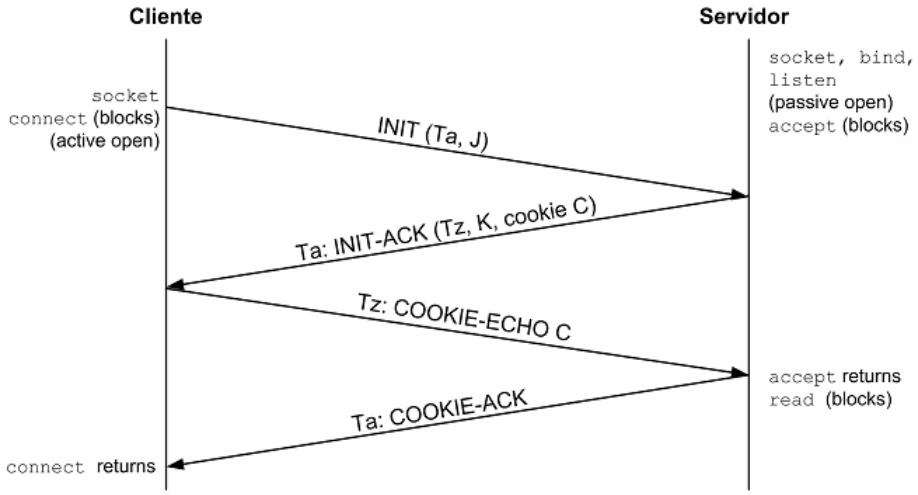


Figura1.13. ESTABLECIMIENTO DE LA COMUNICACIÓN SCTP[7]

(Fuente: Do Carmo G, Peer-to-peer com a utilização do SCTP para aplicativos de Compartilhamento de arquivos, Master Thesis, 2006)

El problema surge cuando los *crackers*[1] intentan dejar fuera de servicio a un servidor haciendo uso del ataque *SYN Flooding,* el cual consiste en forzar los paquetes IP y falsificar la dirección IP origen, y luego inundar un servidor con paquetes SYN de TCP. Entonces el servidor designa recursos para esa conexión y envía un SYN-ACK por cada conexión, esperando un ACK. Usualmente los *crackers* utiliza cientos y hasta millares de computadoras, a las cuales les instalan un agente a cada una, y pueden acceder a estos agentes remotamente, además de manejarlos en conjunto y con ello simultáneamente inundar un servidor, hasta que eventualmente el servidor se llena de conexiones y se vuelve inaccesible para una nueva conexión.[8]

[1] *Cracker,* Persona con altos conocimientos informáticos que intenta acceder a un sistema informático sin autorización, a menudo son con malas intenciones, disponen de muchos medios para introducirse en el sistema.

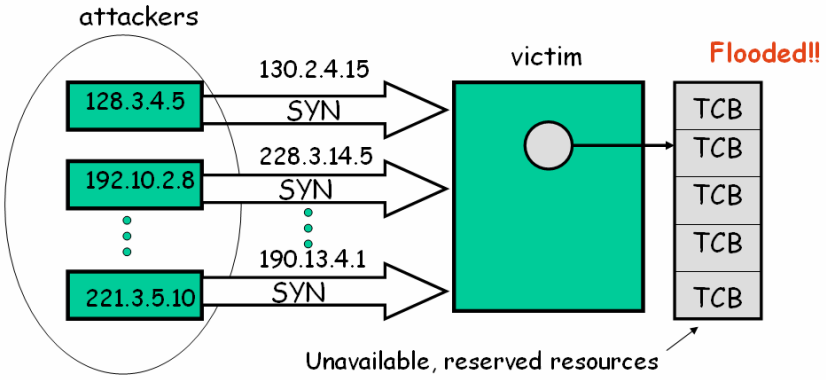


Figura1.14. ATAQUE DE INUNDACIÓN CON SEGMENTOS SYN [8]

Veamos la figura 1.14 donde un conjunto de clientes falsos atacan a un mismo servidor hasta agotar los recursos, entonces cuando un cliente genuino desea abrir una nueva conexión, el servidor se encuentra inhabilitado y ya no le quedan TCB (*Transmission Control Block*) [2].

El protocolo SCTP protege este tipo de ataques, ya que utiliza cuatro mensajes, introduciendo el mensaje de tipo *cookie.* En SCTP, un cliente inicia la asociación con un mensaje INIT, el servidor responde con un mensaje *INIT-ACK*, en el cual se incluye la *cookie* (el cual es solamente utilizado para identificar la asociación). Luego el cliente responde con un *COOKIE-ECHO*, el cual contiene el *cookie* enviado por el servidor. Y es en este punto donde recién el servidor reserva recursos para la asociación y envía un acuse de recibo mediante un mensaje *COOKIE-ACK.*

Además SCTP permite que se añadan datos en los mensajes *COOKIE-ECHO* y *COOKIE-ACK*, para solucionar el retraso de usar cuatro mensajes para el inicio de la asociación en lugar de tres mensajes como lo hace el protocolo TCP.[8][14]

- ***Message framing*,** encuadrar mensajes es una funcionalidad con la cual los mensajes que son comunicados a través del *socket* son preservados, esto quiere decir que si un cliente envía 200 *bytes* a un servidor y luego 100 *bytes*, el servidor leerá los 200 *bytes* y los 100 *bytes* respectivamente, en dos tiempos. El protocolo UDP también trabaja de esta manera, lo que se

[2] TCB (Transmission Control Block): es una estructura que contiene información sobre la conexión.

considera una ventaja para los protocolos que trabajen orientado a los mensajes.[6][14]

Los protocolos UDP y SCTP son también conocidos como orientados al flujo (*stream-oriented*) y el protocolo TCP como orientado a los *bytes* (*byte-oriented*), por lo que TCP no hace uso de esta funcionalidad, de manera que el servidor recibiría un poco más o un poco menos de lo que se envío ya que se realizara una sola lectura. En el caso que se recibe menos, es por que habrá un segunda lectura, en el siguiente mensaje. Además si una aplicación es orientada a los mensajes y trabaja sobre TCP tendrá que implementar un buffer de datos y *message framing* en la capa de aplicación. Veamos la figura 1.15 para comprender mejor esta funcionalidad. El protocolo SCTP provee *Message framing* en la transferencia de datos. Cuando un extremo envía un mensaje, SCTP garantiza que el tamaño del paquete enviado será el mismo que el tamaño del paquete leído por el otro extremo (*endpoint*) [3].[6][2][14]

Para aplicaciones que son orientadas al flujo, como son audio y video, el hecho que no haya un buen encuadramiento es aceptable.

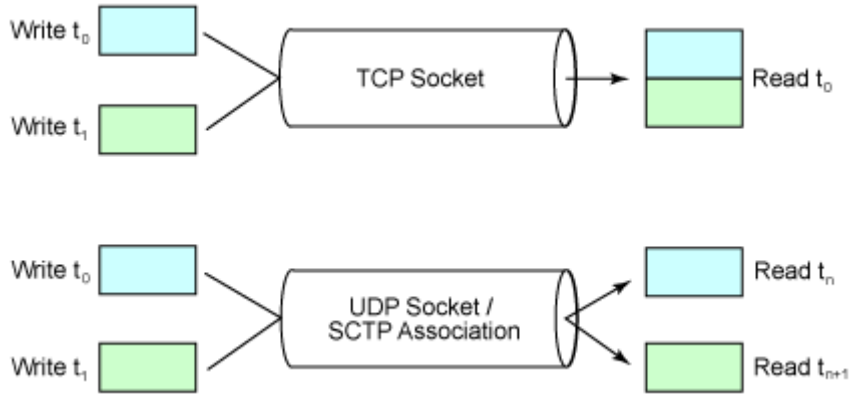


Figura1.15. MESSAGE FRAMING EN UDP/SCTP VS. TCP [6]

[3] Endpoint, En SCTP a los terminales de la asociación se les llama *endpoints.*

- ***Configurable unordered delivery*****,** el protocolo SCTP se puede configurar para un envío desordenado de los datos conservando la confiabilidad, a diferencia del protocolo TCP donde solo se puede enviar de manera ordenada. UDP no garantiza un orden ni confiabilidad. Esta funcionalidad es muy útil en protocolos orientados a los mensajes, en los que los pedidos son independientes y el orden de los mensajes no es importante. En síntesis, con SCTP uno puede configurar para que el envío de paquetes sea desordenado entre flujo y flujo en una misma asociación.[6][14]

- ***Graceful shutdown*****,** podemos decir que TCP y SCTP son protocolos orientados a la conexión considerando que necesitan de un proceso para inicializar la comunicación y otro proceso para terminar la comunicación, por lo tanto UDP es un protocolo no orientado a la conexión, ya que no realiza dichos procesos. La diferencia del *socket* de término de la conexión en SCTP, *shutdown*, es que elimina el término de conexión a medias que realizaba el protocolo TCP, en la cual si uno de los extremos enviaba un mensaje FIN este dejaba de enviar segmentos, y aún podía seguir recibiendo segmentos, hasta que el otro extremo también termine la conexión. Las aplicaciones raramente usan esta funcionalidad, por ello el protocolo SCTP reemplaza esta funcionalidad, con un término de conexión limpio, en donde desde el momento que se envía el mensaje SHUTDOWN, se deja de transmitir en ambas direcciones.[6][7][14]

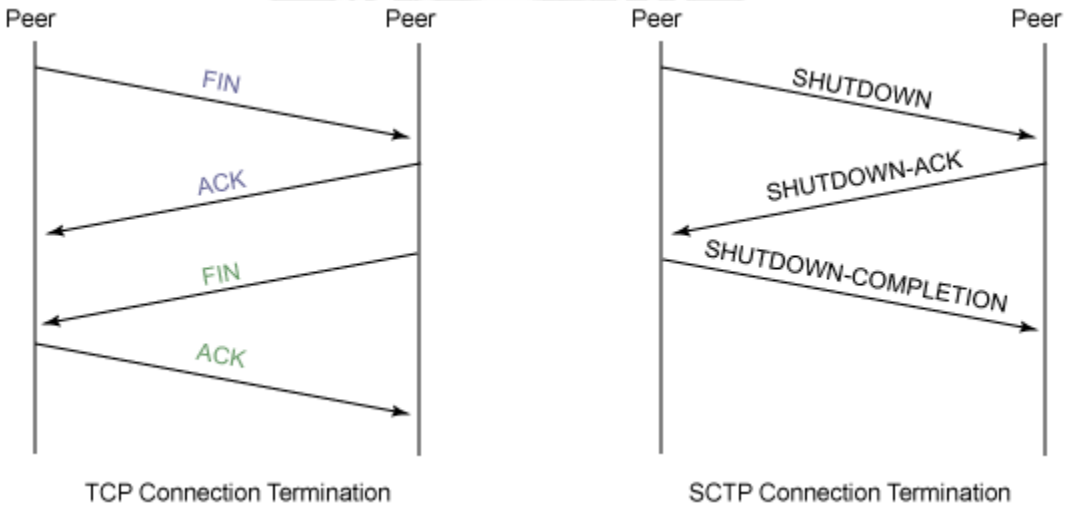


Figura1.16. SECUENCIA DE MENSAJES PARA EL TÉRMINO DE LA COMUNICACION EN TCP Y SCTP. [6]

Como se puede apreciar en la figura 1.17, el protocolo SCTP, termina de enviar los datos antes de dar por terminada la asociación, sin dar lugar al *half-closed* de TCP. Notemos que uno de los *endpoints* envía toda la data antes de enviar el mensaje SHUTDOWN, en forma equivalente el otro *endpoint* envía su data restante antes de responder con el acuse de recibo.[8][14]

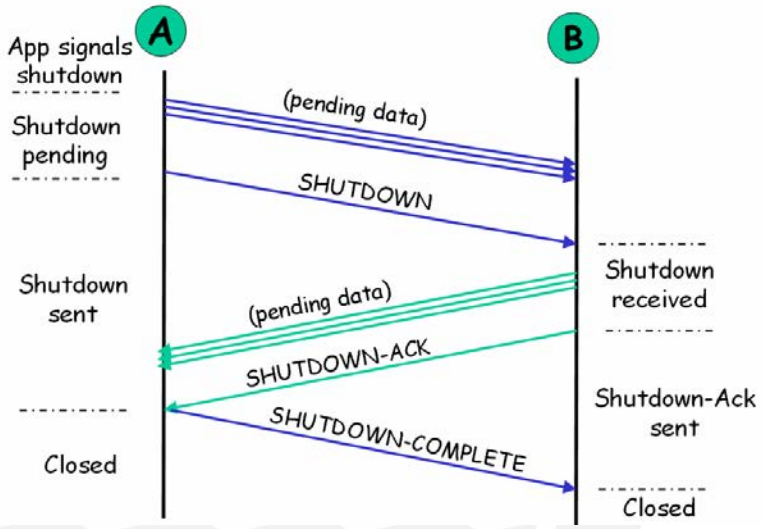


Figura1.17. *GRACEFUL SHUTDOWN[8]*

1.4 Comparación de UDP, TCP y SCTP

Para efectuar una comparación entre estos tres protocolos, primero se mostrará las limitaciones de UDP y las limitaciones de TCP. Seguidamente se mostrará las semejanzas y diferencias entre los protocolos TCP y SCTP, que son los protocolos que realizan un control en el flujo de paquetes.

1.4.1 Limitaciones de UDP

UDP es un protocolo simple ya que ofrece los servicios mínimos de la capa de transporte, a partir de ello podemos hablar de algunas de las limitaciones de UDP.[7]

UDP no ofrece una transferencia confiable ni ordenada de los datos, de manera que una aplicación que trabaje sobre UDP no tendrá la certeza de que los datagramas enviados son recibos por el otro extremo, tampoco tendrá conocimiento del orden en que los datagramas fueron recibidos.[7]

UDP no implementa mecanismo de control de flujo. Por lo cual una aplicación puede desbordar el buffer de recepción del destinatario.[7]

UDP no implementa control de congestión. El hecho de que no haya este control hace que cuando se realiza una congestión, los datos enviados sobre UDP afecten los datos enviados por otras aplicaciones, siendo descartados con mayor probabilidad.[7]

Por estas limitaciones UDP no ofrece un nivel de confiabilidad necesario a algunas aplicaciones ya que es un protocolo orientado a los mensajes. En general UDP puede ser considerado como un protocolo que introduce poco *overhead* [4]. En el caso que una aplicación necesite de un protocolo orientado a los mensajes y a la vez necesite de confiabilidad es decir de un control, si usa UDP tendrá que implementar sus propios mecanismos de control, servicios de retransmisión de mensajes perdidos, corrección del orden de los mensajes, duplicado de mensajes, detección y soluciones de congestionamiento en la red. Sin embargo, la implementación de estos servicios no será la mejor solución si la aplicación es compleja.[7]

[4] Overhead - Es cuando la cabecera tiene un tamaño mayor al que debería tener.

1.4.2 Limitaciones de TCP

Como antes se ha mencionado, el protocolo TCP es orientado a la conexión, ofrece confiabilidad y entrega ordenada de los datos. No obstante, puede presentar algunas limitaciones.

TCP ofrece un orden estricto en la entrega de los datos, y algunas aplicaciones no necesitan de este servicio, ya que solo necesitan de un orden parcial, en este tipo de aplicaciones el hecho de enviar los datos ordenadamente puede ocasionar atrasos no deseables. La perdida de un segmento TCP puede bloquear la entrega de todos los segmente subsecuentes hasta que se retransmita el segmento perdido, a este fenómeno se le conoce como HOL *head-of-line bloking,* el cual será detallado más adelante.[7][14]

Muchas aplicaciones necesitan de la entrega confiable de “mensajes”, TCP es un protocolo que controla la transmisión en base a *bytes*. Algunas aplicaciones orientadas a mensajes logran esta confiabilidad limitando los flujos de *bytes*, o adicionando sus propios registros de límites.[7]

TCP no posee soporte para *host* que tienen múltiples interfaces, también llamados *multihomed host*.[7][14]

TCP es conocido por ser relativamente vulnerable a los ataques DoS *Denial of Service,* por medio de inundación de segmentos SYN.[14]

Con estas limitaciones, el protocolo TCP no es el adecuado para cierto tipo de aplicaciones que son orientados a los mensajes y que no necesitan de un orden estricto en la entrega de datos.[7]

1.4.3 SCTP vs. TCP vs. UDP

El protocolo SCTP fue creado para solucionar las deficiencias y limitaciones de TCP y UDP, fue desarrollado directamente para el transporte de mensajes en la PSTN (*Public Switched Telephone Network*) sobre redes IP. Las aplicaciones que necesiten funcionalidades iguales o semejantes a las de la PSTN pueden utilizar este protocolo. [7][14]

Con UDP se transmiten paquetes, con TCP se transmiten segmentos, con SCTP se transmiten mensajes. [7]

En UDP no se crea una conexión, en TCP se crea una conexión entre ambos terminales, en SCTP se crean asociaciones.

El protocolo UDP es conocido como no orientado a la conexión, orientado a la transacción y también esta dentro de los protocolos orientados a los mensajes. El protocolo TCP es conocido como orientado a la conexión. El protocolo SCTP es orientado a la conexión, y también orientado a los mensajes.[7]

La tabla 1.2 nos muestra un resumen de las características y servicios de TCP, UDP y SCTP.

Tabla 1.2 COMPARACION DE LAS CARACTERISTICAS DE SCTP, TCP Y UDP

CARACTERISTICAS DE PROTOCOLO	**SCTP**	**TCP**	**UDP**
Estado almacenados en los terminales	Si	Si	No
Transferencia confiable de los datos	Si	Si	No
Control de congestion	Si	Si	No
Delimitación de límites en los mensajes	Si	No	Si
Fragmentación e integración de la información	Si	Si	No
Multiplexación de información del paquete	Si	Si	No
Soporte de *multi-homing*	Si	No	No
Soporte de *multi-streaming*	Si	No	No
Envío de datos desordenado	Si	No	Si
Cookie de seguridad para evitar ataques de inundación de SYN	Si	No	No
Mensaje *heartbeat*	Si	No	No

(Fuente[14]: Stewart R. and Xie Q., Stream Control Transmission Protocol(SCTP): A Reference Guide, Addison Wesley, 2002)

A partir de la tabla 1.2 podemos observar que el protocolo TCP se parece más al protocolo SCTP, que el protocolo UDP. Y dado que ambos protocolos TCP y SCTP son orientados a la conexión presentan propósitos parecidos, algunas semejanzas y diferencias son descritas a continuación.

1.4.3.1 Semejanzas entre TCP y SCTP

- **Inicio de la comunicación:** Ambos protocolos pasan por una fase de intercambio de mensajes para establecer el formato de los mismos. Esta fase tiene el mismo propósito en ambos protocolos, que es dar inicio a la conexión o a la asociación.[6][7][14]

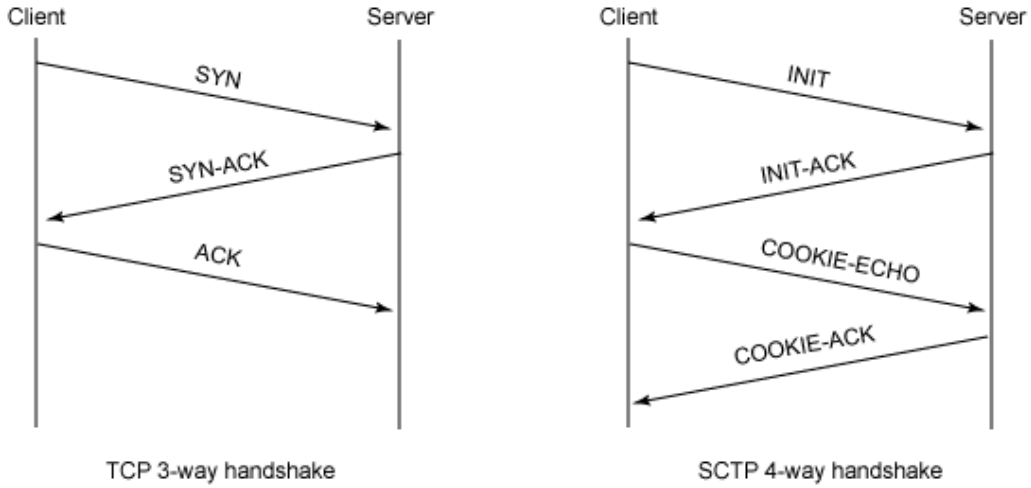


Figura1.18. ESTABLECIMIENTO DE LA COMUNICACIÓN EN TCP Y SCTP[6]

- **Confiabilidad:** Tanto TCP como SCTP ofrecen mecanismo para el envío de datos de manera confiable sobre redes IP.[7][14]

- **Ordenación:** Ambos protocolos ofrecen la certeza de que los mensajes son enviados en orden. Algunas aplicaciones necesitan que sus mensajes sean enviados en el orden correcto, tal es el caso de aplicaciones que realizan monitoreo de equipos, si los mensajes llegan en desorden cuando una falla ocurre, y el mensaje que informa el buen estado llega después podría causar un dato errado.[7][14]

- **Control de congestionamiento:** Tanto TCP y SCTP ofrecen el mismo mecanismo de control de congestionamiento basado en el algoritmo AIMD (*Additive Increase/Multiplicative Decrease*)[5]. Sin este control, las redes de Internet podrían caer por el exceso de tráfico entre los sistemas finales. Y como ambos protocolos usan el mismo algoritmo de congestión se

[5] AIMD (Additive Increase/Multiplicative Decrease), representa un incremento lineal de la ventana de transmisión, y cuando hay una congestión disminuye exponencialmente dicha ventana.

garantiza una justa competencia por el ancho de banda, cuando trabajan sobre una misma red.[7][14]

- **Fin de comunicación:** Ambos protocolos ofrecen una forma d terminar la comunicación.[7][14]

1.4.3.2 Diferencias entre TCP y SCTP

- **Diferencia en el inicio de la comunicación:** Como antes se mencionó, TCP y SCTP ofrecen un mecanismo para iniciar la comunicación punto a punto. Sin embargo existe una diferencia entre el inicio de la misma, y es que TCP utiliza tres mensajes, y SCTP utiliza cuatro mensajes; la razón para utilizar los cuatro mensajes es para proteger contra ataques de SYN *flooding.*[7][14]

- ***Head-of-line bloking* (HOL):** Una diferencia bastante importante es que el protocolo SCTP evita el HOL. El protocolo TCP como antes se mencionó presenta el problema del HOL, y es inherente a la forma de funcionamiento del mismo. Esto se da porque el protocolo TCP transmite los datos en forma ordenada en un solo flujo en una comunicación *full-duplex*, además el protocolo TCP maneja un buffer del cual todos los mensajes deben ser leídos en forma ordenada, de manera que si se pierde un mensaje, todos los mensajes subsecuentes a este permanecen en el buffer hasta que se retransmita el mensaje perdido, a este problema se le conoce como *head-of-line blocking.* Para entender esto imaginemos que la comunicación en TCP es como un caño en cada extremo de los cuales el agua fluye de un lado al otro, esto representa un solo flujo de información en cada dirección. [7][10][14]Ver figura 1.19

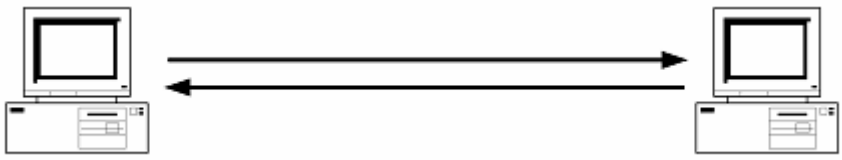


Figura1.19. UNA CONEXÍON EN TCP[7]

En cambio el protocolo SCTP tiene varias caños de agua en cada extremo, esto representa varios flujos de información en cada dirección.[7] Ver figura 1.20

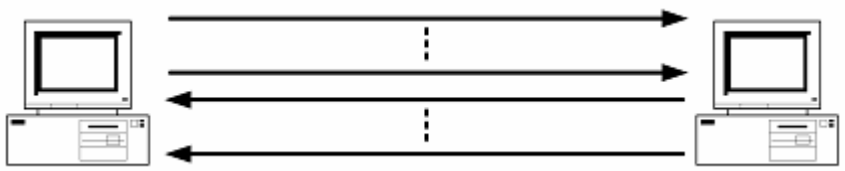


Figura1.20. UNA ASOCIACIÓN EN SCTP[7]

Los mensajes que son enviados en el flujo de TCP tienen que respetar un orden. Los mensajes que van por un mismo flujo en SCTP también respetan un orden, pero los mensajes en distintos flujos son independientes entre sí, por lo que un mensaje pueden llegar de manera desordenada. Además de ello SCTP puede configurar el bit U (*Unorden*) para que los mensajes en el mismo flujo puedan ser leídos en desorden e incluso para la lectura de los mensajes en distintos flujos. Ver la figura 1.21 para ver como actuarían los protocolos TCP y SCTP en caso de la pérdida de un paquete.[7][10][14]

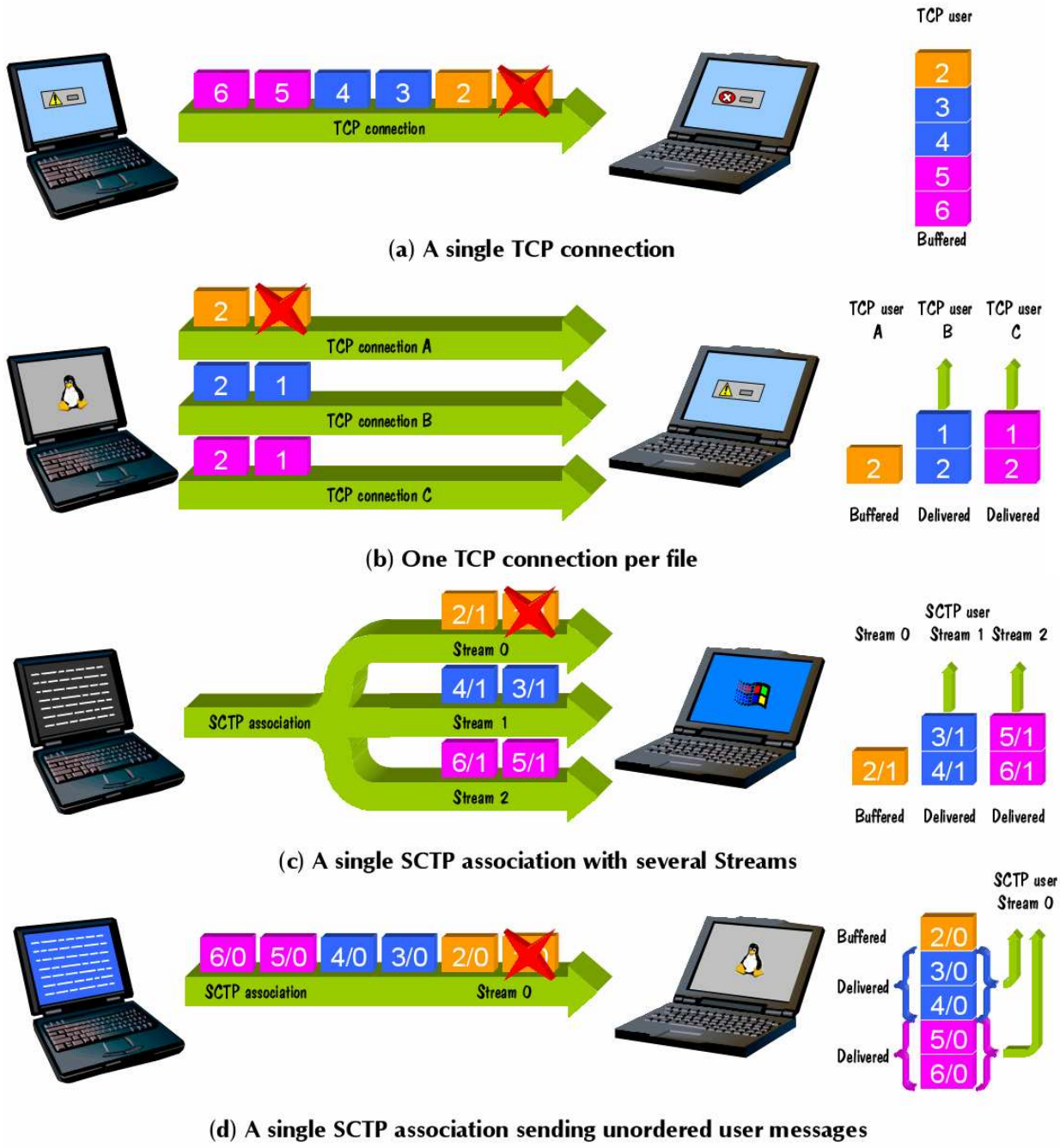


Figura1.21. EL PROBLEMA " HEAD-OF-LINE BLOCKING "[10]

(Fuente: Rodriguez A.,The design of a new reliable transport protocol for IP networks, Master Thesis, 2002)

En todos los casos se envían varios archivos –considerando archivos independientes –, cada archivo tiene dos fragmentos, y esta representado por un color distinto. En el caso (a), se muestra una conexión TCP, donde se pierde el primer fragmento del primer archivo, por lo que esto ocasiona un gran retraso en la entrega de los demás archivos, ya que ninguno de estos podrá ser leído hasta que se retransmita el fragmento perdido. [10][14]

En el caso (b) se muestra una conexión TCP por cada archivo que se desea transmitir, en este caso, al perderse el primer fragmento del primer archivo, no afectará a los demás archivos, ya que cada conexión es independiente de las demás. El hecho de usar varias conexiones implica una implementación en la capa de aplicación adicional para realizar, es decir la aplicación deberá poder manejar hilos. [10][14]

En el caso (c), se muestra una asociación SCTP, la cual cuenta con tres flujos de datos (*streams*), en cada *stream* se transmite un archivo, y ya que los flujos son independientes, la pérdida del fragmento no afecta la comunicación en los demás flujos, y se transmiten en una sola asociación. Y en comparación con el caso (b), no requiere de una implementación adicional en la capa de aplicación, ya que toda la transacción se realiza en la capa de transporte.[10][14]

En el caso (d) se muestra una asociación SCTP que manejas un solo flujo (*stream*), en este caso consideramos que se ha configurado el protocolo para realizar una transferencia desordenada, por ende, al perderse el primer fragmento, los archivos consecuentes son leídos, sin esperar la retransmisión.[10][14]

- **Límites de mensajes:** Los datos transportados en TCP a través de una conexión son considerados como un único flujo de *bytes.* SCTP permite identificar los límites de los mensajes dentro del flujo de bytes, lo cual es beneficioso para las aplicaciones que son orientadas a los mensajes.[7][14]

- ***Multi-homing*:** Otra diferencia muy importante es la implementación de *multi-homing* en el protocolo SCTP, la cual permite que se establezca una asociación utilizando todas las interfaces de red de un *host*.[7][14]

- **Fin de comunicación:** El protocolo SCTP no permite el estado del *half-closed*, en el cual uno de los extremos puede continuar enviando datos.[7][14]

Después de mostrar estas características notamos que el protocolo SCTP ofrece mayores ventajas, sobre el protocolo TCP, siendo las más destacadas la posibilidad de transmitir en múltiples flujos y la entrega desordenada de mensajes en una misma asociación, lo cual puede es utilizado para evitar el HOL que normalmente padece TCP.

CAPITULO II PROGRAMACIÓN DE SOCKETS EN LENGUAJE C SOBRE LINUX

Cuando deseamos comunicarnos por ejemplo cuando enviamos una carta a otra persona en otra ciudad, necesitamos conocer algunos datos, como son el nombre de destinatario, y su dirección. En el caso de la carta si esta es enviada de parte de “Alice” para “Bob” debemos de conocer los nombres del emisor –uno mismo- y del destinatario; las direcciones de ambos; y de un medio de comunicación el cual puede ser US *Postal Service,* con estos tres parámetros la comunicación se puede realizar. [9] Ver la figura 2.1.

Figura2.1. ENVIO DE UN CARTA[9]

(Fuente[9]: Borisov N, Introduction to UNIX Network Programming, 2006)

Para poder realizar esta comunicación; Alice debe escribir la dirección de Bob en la carta; debe poner el nombre de Bob, por si hay personas que comparten el mismo buzón de correo; debe pegarle una postal, pues es necesario pagar por el envío; y por último Alice debe poner su nombre y su dirección, por si Bob desea devolver el mensaje. Luego el mensaje es enviado a través de US *Postal Service*. Entonces Bob, por su parte, debe la instalar un buzón de correo, para que la gente sepa y tenga un lugar donde enviarle las cartas, recibe la carta, rompe el sobre y lee el mensaje.[9]

La comunicación en TCP/IP es muy similar al envío de la carta de Alice para Bob, solo que se habla una diferente terminología, y la tecnología es distinta, la cual se supone que debe ser mejor, más rápida, más confiable, y más barata.[9]

Si ahora Alice y Bob se quiere comunicar sobre TCP/IP entonces se deben formar dos simples programas (Ver figura 2.2), análogamente al envío de la carta, Alice y Bob deben tener una dirección, solo que esta vez debe ser una dirección IP; en lugar de los nombres ahora manejaran un puertos, y esto es porque múltiples programas comparten la misma dirección IP, así como varias personas en mismo vecindario podrían compartir un buzón de correo; Alice debe de implementar un proceso de envío del mensaje, y Bob debe de implementar un proceso para recibir el mensaje, similar al proceso que ellos mismo hacían para enviar y recibir la carta. El término análogo al buzón de correo es el *socket* –esto se detallara más adelante – y el medio de comunicación es la red de Internet.[9]

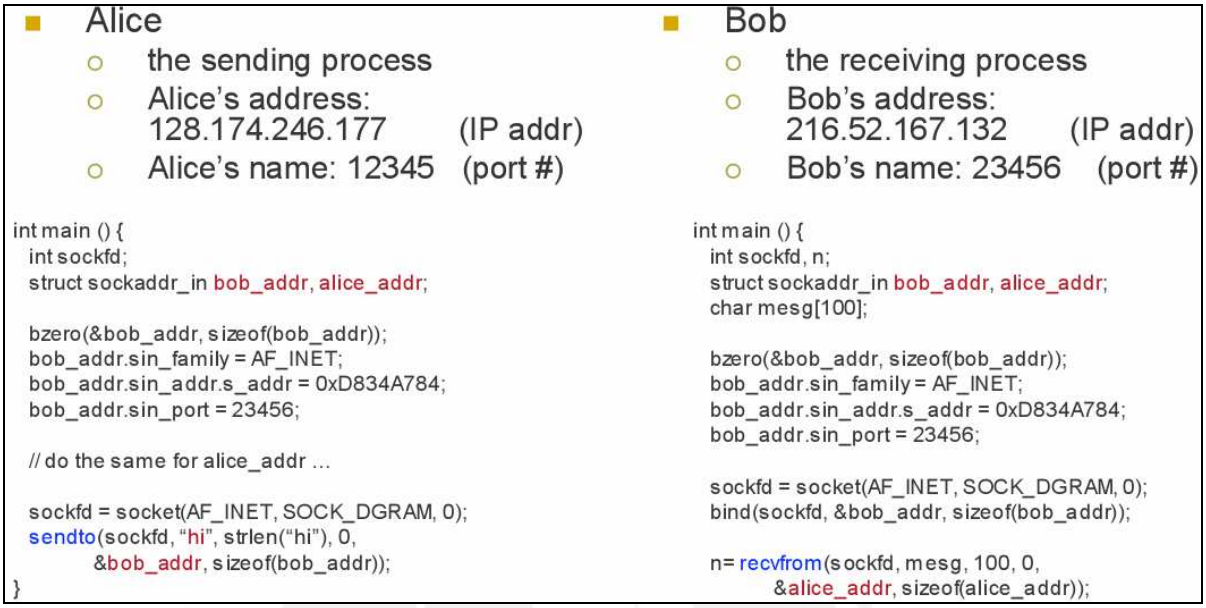


Figura2.2. DOS SIMPLES PROGRAMAS DE NETWORKING[9]

2.1 El *SOCKET*

Un *socket* es un método de comunicación entre un programa cliente y un programa servidor en una red, es el punto final en una comunicación, en el ejemplo anterior era similar al buzón de correo. [9]

Un *socket* queda definido por una dirección IP, un protocolo, y un puerto. En si un *socket* es un par de ficheros uno en el *host* cliente y el otro en el *host* servidor, los cuales son usados para que la aplicación servidor y la aplicación cliente lean y escriban la información que será transmitida a través de la red. Los *sockets* son utilizados en base a un sistema de peticiones o de *llamadas de función*, la cual es llamada API (*Application Programming Interface*) de los *sockets.* [2][9]

También podemos decir que un *socket* es la forma en que las computadoras se comunican, haciendo uso de descriptores de archivos estándar de *UNIX.* En Internet hay varios tipos de *sockets* pero nos centraremos en explicar solo tres: los *sockets* de flujo (*SOCK_STREAM*), los *sockets* de datagramas (*SOCK_DGRAM*) y los *sockets* de datagramas secuenciales (*SOCK_SEQPAQUET*)[2][9]

2.1.1 Tipos de *Sockets*

- ***Sockets* de Flujo *SOCK_STREAM***

Los *sockets* de flujo están libres de errores, si enviamos al socket de flujo tres objetos a, b y c entonces llegaran al destino en el mismo orden a-b-c. Este tipo de *socket* es usado por el protocolo TCP, ya que se transmiten los mensajes ordenadamente.[2]

- ***Sockets* de Datagrama *SOCK_DGRAM***

Este tipo de *sockets* son usados por el protocolo UDP, no necesitan de conexión, se construyen los paquetes con la información de un destino y son enviados.[2]

- ***Sockets* de Paquetes Secuenciales *SOCK_SEQPAQUET***

Este tipo de sockets es únicamente usado por el protocolo SCTP

El protocolo SCTP ofrece dos formas de trabajo, la primera es llamada TCP-*Style* [6] y la segunda UDP-*Style* [7] los cuales serán detallados más adelante.[2][14]

2.1.2 Las estructuras en los Sockets

La programación de estructuras en el lenguaje C es utilizada en la implementación de los *sockets.* Una estructura es una agrupación de variables, es un tipo de dado que puede contener otros tipos de datos; las estructuras en la programación de *sockets* son usadas para almacenar información de direcciones.[2]

```
struct sockaddr {
  uint8_t      sa_len;
  sa_family_t  sa_family;    /* address family: AF_xxx value */
  char         sa_data[14];  /* protocol-specific address */
};
//Fuente: Stevens R, UNIX Network Programming, Third Edition, 2003
```

Por ejemplo la estructura *struct sockaddr* contiene información del socket, como la familia y la dirección del protocolo.[2]

---

[6] TCP-*Style:* En algunos documentos es conocida como one-to-one socket.

[7] UDP-*Style*: En algunos documentos es conocido como one-to-many sockets.

La estructura *struct sockaddr_in* hace referencia a los elementos del *socket* ; la familia de la dirección, el puerto y la dirección de Internet.[2]

```
Struct sockaddr_in {
  uint8_t         sin_len;      /* length of structure (16) */
  sa_family_t     sin_family;   /* AF_INET */
  in_port_t       sin_port;     /* 16-bit TCP or UDP port number */
                                /* network byte ordered */
  struct in_addr  sin_addr;     /* 32-bit IPv4 address */
                                /* network byte ordered */
  char            sin_zero[8];  /* unused */
};
//Fuente: Stevens R, UNIX Network Programming, Third Edition, 2003
```

La estructura *struct in_*addr no es muy utilizada, sin embargo, esta definida como una unión.

```
Struct in_addr {
  in_addr_t   s_addr;           /* 32-bit IPv4 address */
                                   /* network byte ordered */
};
//Fuente: Stevens R, UNIX Network Programming, Third Edition,
2003
```

2.1.3 Las conversiones en los *Sockets*

Hay dos formas de almacenar los octetos en la memoria, si primero se almacena el LSB[8] se llama *little-endian byte order*; en caso contrario cuando primero se almacena el MSB[9] se llama *big-endian byte order* este último es también conocido como "Ordenación de *bytes* para redes", ya que en la lectura de las direcciones IP es más eficiente leer y escribir el MSB. Un sistema de conversiones es necesario por que algunos sistemas utilizan estos sistemas para guardar sus datos internamente., por ejemplo las tarjeras madre de IBM y los procesadores de Motorota utilizan *big-endian byte order*, los procesadores de Intel utilizan *little-endian byte order*. En la figura 2.3 se muestra gráficamente estos dos sistemas de almacenamiento.[2][5]

[8] LSB: *least significant bit*, bit menos significativo.

[9] MSB: *most significant bit*, bit más significativo.

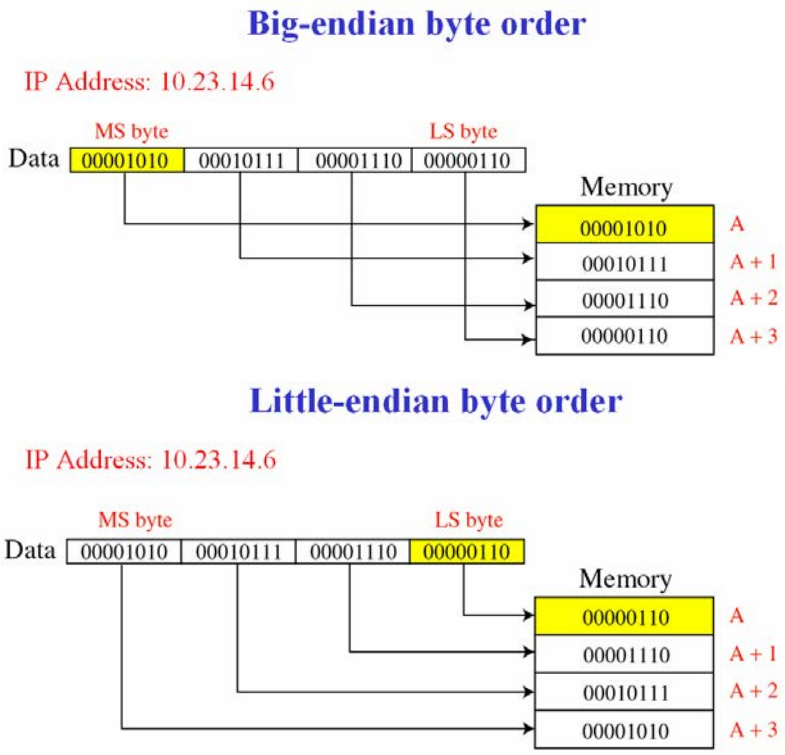


Figura2.3. BIG-ENDIAN BYTE ORDER y LITTLE-ENDIAN BYTE ORDER.[5]

La forma en que se utiliza un sistema de conversiones en sockets para los tipos de ordenamiento antes mencionados, es usando las siguientes funciones. Debemos considerar que hay dos tipos de variables a las que podemos convertir *short* y *long.*[2][5]

- htons()→ ``*Host to Network short*'', Nodo a variable corta de Red
- htonl()→ ``*Host to Network long*'', Nodo a variable larga de Red
- ntohs()→ ``*Network to Host short*'', Red a variable corta de Nodo.
- ntohl()→ ``*Network to Host long*'' , Red a variable larga de Nodo''

Estas funciones pueden cambiar de orden una variable de dos bytes como es el valor de un puerto, o una variable de cuatro bytes como es una dirección IP, veamos la figura 2.3, la cual muestra como son las cambios dependiendo si es un puerto o una dirección IP.[2][5]

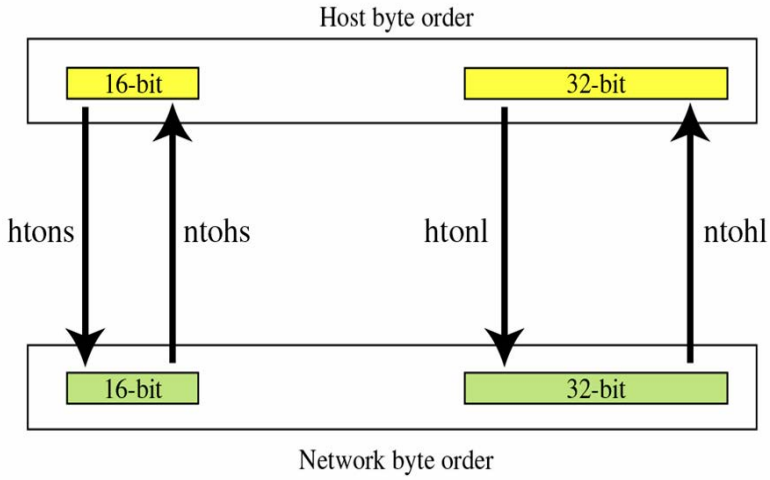


Figura2.4. TRANSFORMACIÓN DE BYTE ORDER[5]

2.1.4 Algunas de las funciones utilizadas en la programación de *Sockets*

- **Función *socket()***

En esta función se define el tipo de protocolo de comunicación que utilizara el *socket*, el tipo de familia a la que pertenece, y el tipo de socket.

```
#include <sys/socket.h>

int socket (int family, int type, int protocol);

            Returns: non-negative descriptor if OK, -1 on error
 //Fuente: Stevens R, UNIX Network Programming, Third Edition,
2003
```

En la figura 2.5 se muestra las opciones que en este caso vamos estudiar, lo cual es lo concerniente para la comunicación de los protocolo TCP, UDP y SCTP. Notemos en la figura las familias para IPv4 e IPv6, notemos también que el protocolo SCTP utiliza dos tipos de *sockets* SOCK_STREAM y SOCK_SEQPAQUET.[2]

La función *socket()* devuelve un descriptor de fichero, el cual es utilizado para llamadas al sistema. Si devuelve -1 entonces ocurrió un error.[2]

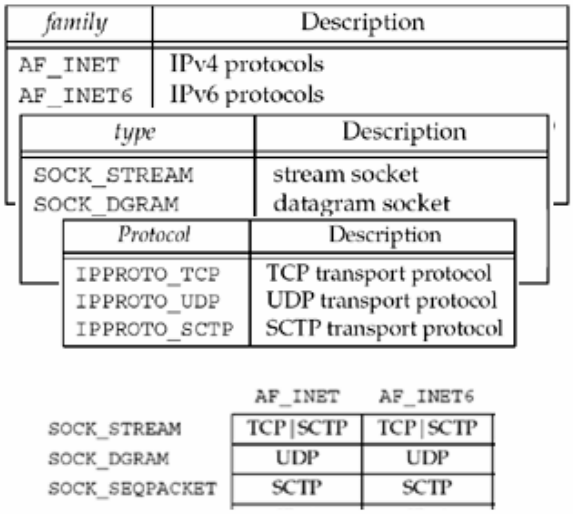


Figura2.5. OPCIONES DE LOS PARAMETROS DE LA FUNCION SOCKET[2]

- **Función *bind()***

La función *bind()* es la encargada de asociar un *socket* con un puerto de un *host*. Devuelve -1 cuando hay un error.

```
#include <sys/socket.h>

int bind (int sockfd, const struct sockaddr *myaddr, socklen_t
addrlen);

                                    Returns: 0 if OK,-1 on error
 //Fuente: Stevens R, UNIX Network Programming, Third Edition,
2003
```

Entre los parámetros de esta función tenemos *sockfd* es el descriptor de fichero el cual debe ser devuelto por la función *socket()*, el parámetro *myaddr* es un puntero a una estructura *sockaddr,* y el parámetro *addrlen* es la longitud de la estructura *sockaddr* a la cual apunta el puntero *myaddr.* Se debe establecer como *sizeof(struct sockaddr)* para asegurar que se obtenga la longitud de la estructura.[2]

- **Función *connect()***

La función *connect()* es usada para realizar la conexión a un puerto definido en una dirección IP. Devuelve -1 cuando hay un error

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *servaddr,
socklen_t addrlen);
                                        Returns: 0 if OK,-1 on error
 //Fuente: Stevens R, UNIX Network Programming, Third Edition,
2003
```

Entre los parámetros de esta función tenemos *sockfd* es el descriptor de fichero el cual debe ser devuelto por la función *socket()*, el parámetro *servaddr* es un puntero a una estructura *sockaddr,* y el parámetro *addrlen* es la longitud de la estructura *sockaddr* a la cual apunta el puntero *myaddr.* Se debe establecer como *sizeof(struct sockaddr)* para asegurar que se obtenga la longitud de la estructura.[2]

- **Función *listen()***

La función *listen()* es usada cuando se esta esperando conexiones entrantes, es decir espera a que alguien intente conectarse con el *host* para luego tomar una acción por lo cual después de la función *listen()* se debe llamar a la función *accept().*

```
#include <sys/socket.h>

int listen (int sockfd, int backlog);
                                        Returns: 0 if OK,-1 on error
 //Fuente: Stevens R, UNIX Network Programming, Third Edition,
2003
```

Entre los parámetros de esta función tenemos *sockfd* es el descriptor de fichero el cual debe ser devuelto por la función *socket()*, y el parámetro *backlog*, que representa el número de conexiones permitidas.[2]

- **Función *accept()***

Luego de que el otro extremo intenta conectarse (usando la función *connect()*) se debe aceptar la conexión, esto se hace mediante la función *accept()* . Esta función cuando es llamada por un servidor TCP, borra el primer pedido de conexión de la cola. Si la cola esta vacía la función *accept()* es puesta a dormir.[2]

```
#include <sys/socket.h>

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t
*addrlen);

            Returns: non-negative descriptor if OK, -1 on error
 //Fuente: Stevens R, UNIX Network Programming, Third Edition,
2003
```

La función *accept(),* al igual que la función *socket(),* devuelve un descriptor de fichero, el cual es utilizado para llamadas al sistema. Si devuelve -1 entonces ocurrió un error.[2]

Entre los parámetros de esta función tenemos *sockfd* es el descriptor de fichero el cual debe ser devuelto por la llamada a la función *listen()*; el parámetro *cliaddr* es un puntero a una estructura *sockaddr_in,* en la cual se debe estar definido qué *host* se esta conectando y desde qué puerto*;* y el parámetro *addrlen* es la longitud de la estructura *cliaddr,* se debe establecer como *sizeof(struct cliaddr)* para asegurar que se obtenga la longitud de la estructura.[2]

2.2 Vista breve del modelo Cliente – Servidor

El modelo cliente – servidor es un estándar de ejecución de las aplicaciones en la red.

Consta de dos extremos, el servidor se ejecuta en uno de los dos extremos y es el que se encarga de gestionar el acceso a algún determinado recurso, en el otro extremo (puede ser el mismo *host*) se ejecuta el cliente, el cual se encarga de realizar peticiones de servicio al servidor. Estas peticiones son originadas por la necesidad de acceder a alguno de los recursos que maneja el servidor.

El servidor se encuentra en espera de peticiones de servicio. Cuando se produce un evento de una petición, el servidor despierta y atiende al cliente. Una vez terminado el servicio, el servidor regresa al estado de espera de peticiones.

Según la forma de prestar los servicios, los servidores se clasifican en servidores interactivos y servidores concurrentes. Los servidores interactivos, reciben la petición de servicio y la atienden, si hay muchas peticiones, el tiempo de espera de las peticiones se prolonga; los servidores concurrentes, reciben las peticiones y crea procesos para poder atenderlas. La ventaja de este último, es que puede recibir las peticiones a muy alta velocidad.[2]

2.3 API de sockets para SCTP

El trabajo de la expansión de *sockets* para el protocolo SCTP tiene tres metas principales:

- Mantener la consistencia con la API (*Application Programming Interface*) existente, lo que implica que los API de *sockets* para SCTP tener consistencia con el API de *sockets* de UDP, TCP, IPv4 e IPv6.[2]

- Soporte de TCP-*style interface*, el propósito de esta meta es brinda un camino fácil para la migración al protocolo SCTP de las aplicaciones existentes que utilizan el protocolo TCP, esta migración se hace mediante sencillos cambios que se debe realizar en el código fuente de las aplicaciones. Sin embargo, con este cambio no se alcanzas las mejoras que ofrece el protocolo SCTP, para lograrlo las aplicaciones tendrán que cambiar su código fuente usando UDP-*style interface.* En algunos documentos TCP-*style interface* es conocido como *one-to-one socket*.[2][14]

- Soporte de UDP-*style interface*, el propósito de esta meta es proveer un método fácil y eficiente, para explotar todas las características del protocolo SCTP, por tanto el UDP-*style interface* es superior al TCP-*style interface.* En algunos documentos UDP-*style interface* es conocido como *one-to-many socket*. [2][14]

2.3.1 Sockets SCTP - TCP-style interface

Para este estilo de programación de SCTP vamos a utilizar el código de un cliente y un servidor programados en TCP, para luego hacer unos sencillos cambios para poder operar con el protocolo TCP. El código fuente de estas aplicaciones se encuentra en el Anexo 1.[2]

Un cliente – servidor programado en TCP tiene el siguiente diagrama de flujo (ver figura 2.6), donde el servidor abre el canal de comunicación con la función *socket();* se asocia al socket con un puerto con la función *bind();* y se pone a disposición de peticiones de servicio con la función *listen();* cuando recibe una petición, si es un servidor interactivo acepta la conexión con la función *accept()* y atiende al cliente, si es un servidor concurrente, creara un proceso *fork* para atender la petición. Con las funciones *read()* y *write()*, se realiza la transferencia de paquetes, con la función *close()* se termina la conexión. El cliente, por su lado, también abre el canal de comunicación, intenta conectarse con la función *connect()* , si se establece la comunicación, realiza la transferencia de datos, y finaliza la conexión con la función *close()* y termina la ejecución.[2]

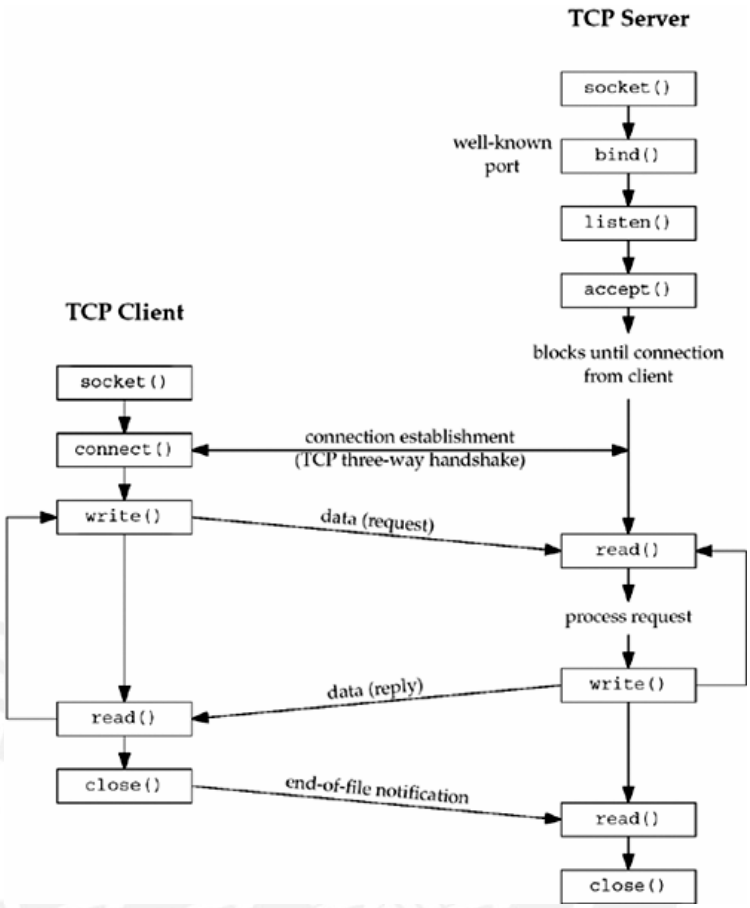


Figura2.6. *DIAGRAMA DE FLUJO CLIENTE-SERVIDOR TCP[2]*

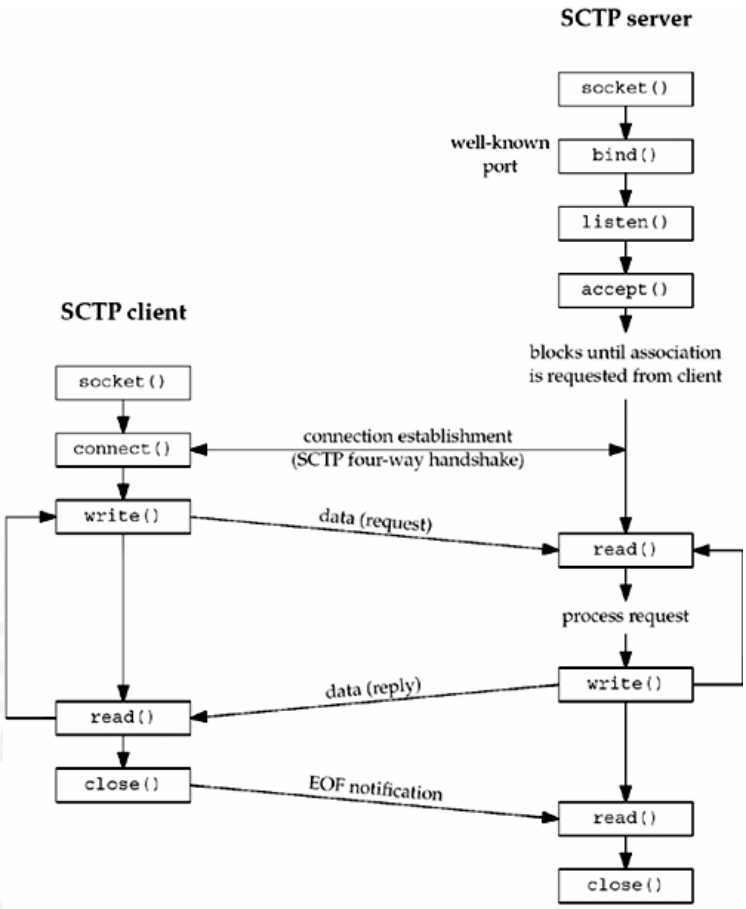


Figura2.7. *DIAGRAMA DE FLUJO CLIENTE-SERVIDOR SCTP[2]*

Ahora veamos un diagrama de flujo cliente – servidor trabajando en TCP-style usando el protocolo SCTP (ver figura 2.7), notemos que es casi idéntico al diagrama de flujo del cliente- servidor TCP, esto es por que solo hace un sencillo cambio, en lugar de enviar segmentos TCP se envían mensajes SCTP.[2]

No obstante, antes de ello debemos aclarar algunas diferencias en las cuales hay que tener cuidado:

1. El cambio más sencillo que debemos hacer es cambiar el parámetro *IPPROTO_TCP* por *IPPROTO_SCTP* cuando llamamos a la función *socket()*.[14]

```
...descriptor_archivo=socket(AF_INET, SOCK_STREAM,
IPPROTO_TCP)...

...descriptor_archivo=socket(AF_INET, SOCK_STREAM,
IPPROTO_SCTP)...
```

2. Otras opciones comunes son TCP_NODELAY y TCP_MAXSEG que deben ser convertidas a SCTP_NODELAY y SCTP_MAXSEG respectivamente.

3. Algunas aplicaciones sobre TCP usan la funcionalidad de *half-close*, en estos casos las aplicaciones deberán ser rescritas, ya que SCTP no implementa dicha funcionalidad.[2][14]

4. SCTP preserva los límites de los mensajes, sin embargo, en la programación de las aplicaciones existentes no se tuvo en cuenta ello, con lo que un mensaje podría ser separado en dos o más mensajes, para que tenga consistencia con la programación orientada a los bytes.[2][14]

Para ver que tan sencillo es migrar de una aplicación hecha sobre TCP a SCTP, usamos el código de un cliente y un servidor propuesto en los ejemplos de "*Stevens R, UNIX Network Programming, Third Edition, 2003*" y le cambiamos el mensaje, en el caso del servidor TCP, el mensaje enviado del servidor al cliente fue "Bienvenido a mi servidor", en el caso del servidor SCTP, el mensaje fue "Bienvenido a mi servidor SCTP".

Luego de ello compilamos y generamos un archivo ejecutable para seguidamente capturar los paquetes que generan estas aplicaciones, véase figura 2.8 y figura 2.9. Con esta sencilla aplicación podemos analizar el tipo de mensajes que utiliza TCP y SCTP para realizar la comunicación, por ejemplo notemos que el protocolo TCP para la inicio de la conexión utiliza tres mensajes, SYN, SYN-ACK, ACK, mientras que el protocolo SCTP utiliza cuatro mensajes que son INIT, INIT_ACK, COOKIE_ECHO, COOKIE_ACK, para el establecimiento de la asociación. Y así podemos analizar cada uno de estos mensajes.

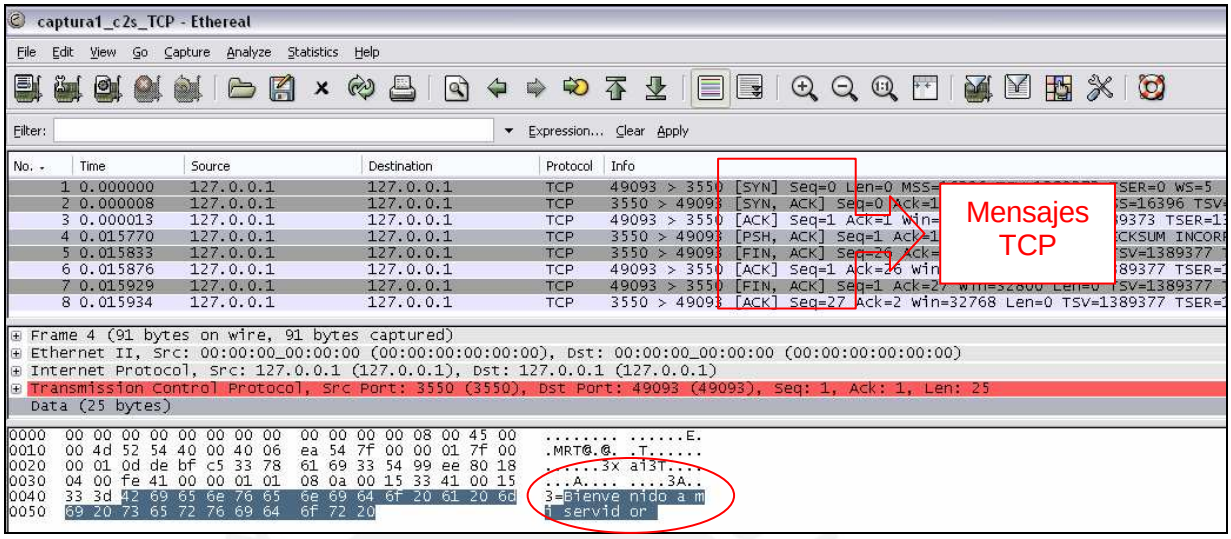


Figura2.8. CAPTURA DE TRAMAS CLIENTE/SERVIDOR SOBRE TCP

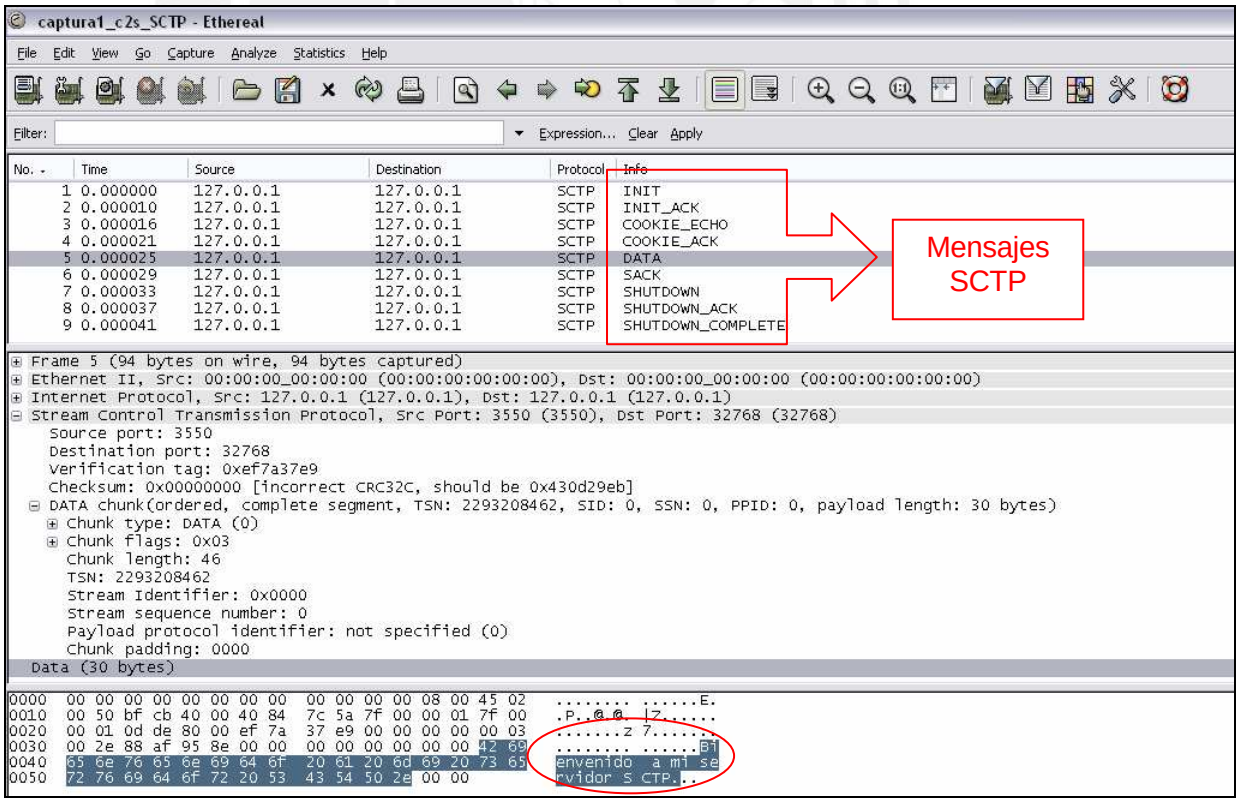


Figura2.9. CAPTURA DE TRAMAS CLIENTE/SERVIDOR SOBRE TCP

2.3.2 Sockets SCTP - UDP-style interface

En este estilo de programación se implementan nuevas funciones propias de SCTP, así como muchas nuevas estructuras.

Por ejemplo se tienen las funciones *sctp_sendto(), sctp_sendmsg() y sctp_recvmsg()* para el intercambio de mensajes.

Aunque podemos utilizar algunas de las funciones clásicas hay algunas que tienen unos cambios para ofrecer las funcionalidades del protocolo SCTP.

Veamos en la figura 2.10 un diagrama de cómo es la comunicación entre programas que usan la interfaz UDP-style.

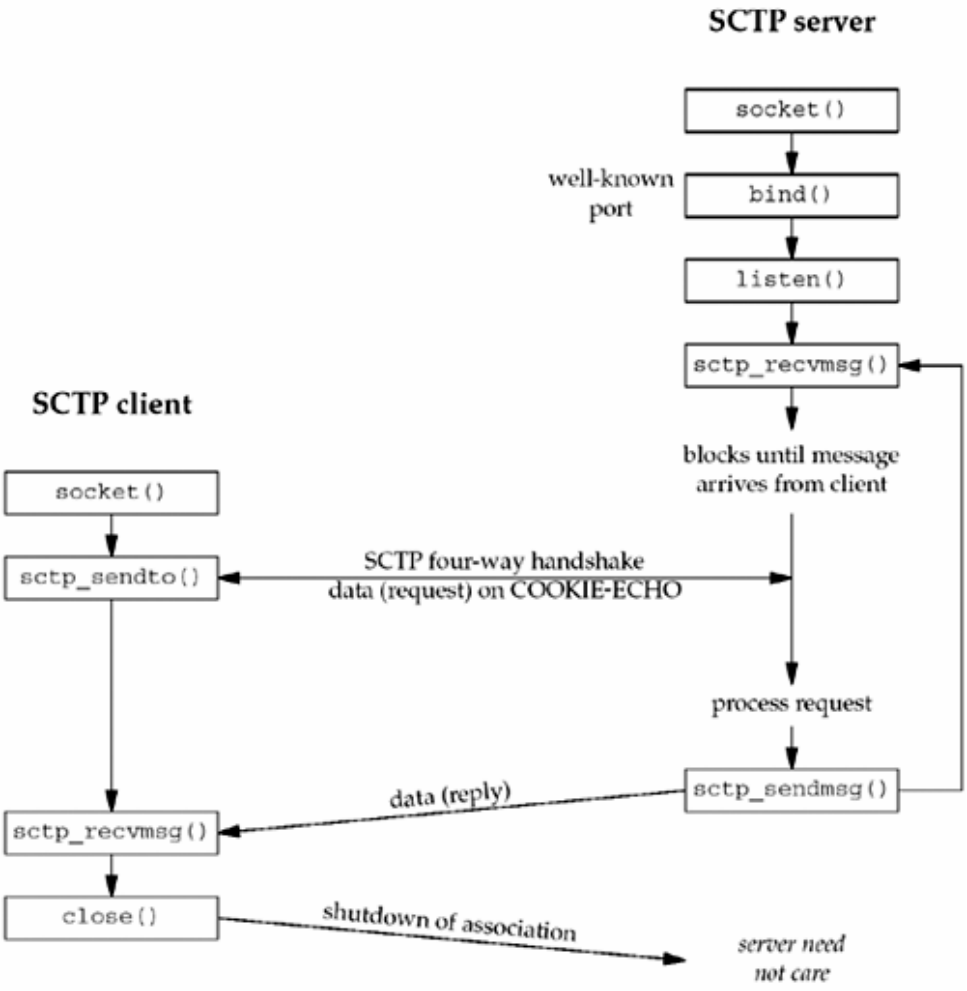


Figura2.10. DIAGRAMA DE FLUJO CLIENTE-SERVIDOR SCTP- UDP STYLE SOCKET[2]

- **Función sctp_bindx*()***

La función *sctp_bindx()*, es similar a la función *bind(),* con la diferencia de que con la función *bind()* solo se puede asociar un *socket* con un puerto de un *host,* en cambio con el protocolo SCTP se puede tener un arreglo de sockets para asociar las diferentes interfaces.

Devuelve -1 cuando hay un error.

```
#include <netinet/sctp.h>

int sctp_bindx(int sockfd, const struct sockaddr *addrs, int
addrcnt, int flags);

                                    Returns: 0 if OK,-1 on error
  //Fuente: Stevens R, UNIX Network Programming, Third Edition,
2003
```

Entre los parámetros de esta función tenemos *sockfd* es el descriptor de fichero el cual debe ser devuelto por la función *socket()*, el segundo parámetro es una lista de paquetes de estructuras de socket IPv4 o socket IPv6. Estos dos tipos de estructuras *sockaddr_in* y *sockaddr_in6* tienen diferentes tamaños, sin embargo no se desperdicia memoria, cuando se usa ambos, solo hay que apuntar bien a la dirección de la estructura. El parámetro *flags* puede ser *SCTP_SOCKOPT_BINDX_ADD* o *SCTP_SOCKOPT_BINDX_REM* con lo que podemos agregar una dirección al la estructura o quitar la dirección respectivamente. El parámetro *addrcnt* indica el número de direcciones IP que se utilizan.[2][23]

- **Función sctp_connectx*()***

La función *sctp_connectx()*, es similar a la función *connect(),* es usada para conectar a un nodo *Multihoming*

Devuelve -1 cuando hay un error.

```
#include <netinet/sctp.h>

int sctp_connectx(int sockfd, const struct sockaddr *addrs, int
addrcnt);
```

```
                                Returns: 0 for success, –1 on error
 //Fuente: Stevens R, UNIX Network Programming, Third Edition,
2003
```

Entre los parámetros de esta función tenemos *sockfd* es el descriptor de fichero el cual debe ser devuelto por la función *socket()*, el segundo parámetro es una lista de paquetes de estructuras de socket IPv4 o socket IPv6. El parámetro *addrcnt* indica el número de direcciones IP que se utilizan.[2][23]

- **Función sctp_sendmsg*()***

Una aplicación puede hacer uso de varias de las características de SCTP haciendo uso de esta función.

```
ssize_t sctp_sendmsg(int sockfd, const void *msg, size_t msgsz,
const struct sockaddr *to, socklen_t tolen, uint32_t ppid,
uint32_t flags, uint16_t stream, uint32_t timetolive, uint32_t
context);
                Returns: the number of bytes written, –1 on error
//Fuente: Stevens R, UNIX Network Programming, Third Edition,
2003
```

Esta función tiene un método de envío simplificado con el costo de más argumentos. El parámetro *sockfd* es el descriptor de fichero el cual debe ser devuelto por la función *socket()*, el segundo parámetro *msg* es un puntero a un *buffer* de tamaño *msgsz*, el cual será enviado de nodo a nodo. El parámetro *tolen* es la dirección donde va ser guardada. El parámetro *ppid* indica el *payload* el cual será entregado con el *chunk* de datos.

El parámetro *flags* es para identificar algunas opciones SCTP como que trabaje con *gracefull shudown,* que los mensajes se envíen de manera desordenada, o que la transmisión sea parcialmente confiable.[2][23]

- **Función sctp_recvmsg*()***

Al igual que la función *sctp_sendmsg*(), la función *sctp_recvmsg*() provee una interface amigable del usuario a las características de SCTP.

Esta función contiene parámetros similares a sctp_sendmsg(), pero además tiene el parámetro *from*, que es una estructura que guarda la dirección del remitente, y el parámetro *msg_flags* con el se pueden enviar opciones del mensaje.[2][23]

```
ssize_t sctp_recvmsg(int sockfd, void *msg, size_t msgsz,
struct sockaddr *from, socklen_t *fromlen, struct
sctp_sndrcvinfo *sinfo, int *msg_flags);

              Returns: the number of bytes written, –1 on error
//Fuente: Stevens R, UNIX Network Programming, Third Edition,
2003
```

- **Ejemplos de código fuente usando UDP-*style***

En el Anexo 1 se adjuntan los códigos de cuatro aplicaciones, dos clientes con sus respectivos servidores; el primer código es *streamcount_echoclient.c* que trabaja con el servidor *streamcount_echoserver.c,* estos dos programas se intercomunican y negocian el número de *streams* que tienen que enviar. Este proceso es importante ya que identifican que cantidad de *streams* esta dispuesto a transmitir cada nodo, siendo el de menor valor quien define el valor. Los otros dos programas *streamsend_echoclient.c* y *streamsend_echoserver.c* hacen lo mismo, pero además realizan un intercambio de mensajes.

Por ejemplo si el servidor tiene disponible 30 *streams* de entrada y 30 *streams* de salida, y el cliente tiene disponible 20 *streams* de entrada y 50 *streams* de salida, entonces al negociar quedaran con 20 *streams* y 30 *streams* respectivamente, veamos en la figura los resultados de este ejemplo.

```
isaac@isaac-laptop:~/workspace/linuxjournal2_sctp$ ./streamsend_echo_server 127.0.0.1 30 30
Negociando numero de streams...
Servidor tiene disponible: 30, output atreams; 30
Se llega al acuerdo:
 input streams: 30, output streams: 20
read 34 bytes on channel 0
sinfo flags: 0
Enviando mensaje SCTP al servidor


isaac@isaac-laptop:~/workspace/linuxjournal2_sctp$ ./streamsend_echo_client 127.0.0.1  20 50 1
Negociando numero de streams...
Cliente tiene disponible: input streams: 20, output streams: 50
Se llega al acuerdo:
 input  streams: 20, output streams: 30
isaac@isaac-laptop:~/workspace/linuxjournal2_sctp$
```

Figura2.11. RESULTADOS CLIENTE-SERVIDOR SCTP- UDP STYLE SOCKET

En la figura 2. 12 podemos apreciar que la negociación del número de *streams* a usar se hace con los dos primeros mensajes *INIT* e *INIT-ACK.*

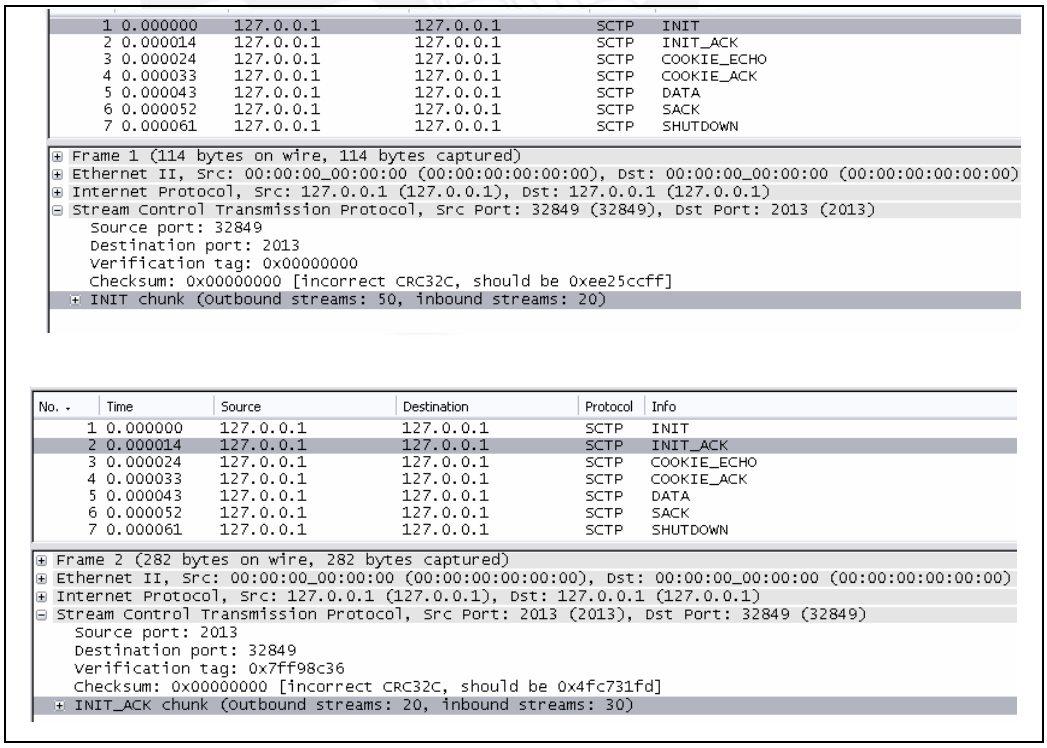


Figura2.12. CAPTURA CLIENTE-SERVIDOR SCTP- UDP STYLE SOCKET

CAPITULO III SELECCIÓN E IMPLEMENTACIÓN DE LA APLICACIÓN

.

En este capítulo se mencionan las herramientas a utilizar, el sistema operativo, el porqué se utilizaron, además se mencionan herramientas que utilizan *sockets* del API de SCTP, las cuales servirán para realizar las pruebas en el capítulo IV.

3.1 Selección de la distribución de Linux y del IDE

Debemos de enfatizar que este trabajo de Tesis es desarrollado en Linux puesto que SCTP esta desarrollado sobre Linux, y no sobre Windows, y esto porque el sistema operativo Windows es de código cerrado y habría que esperar que se implemente estas librerías en el *kernel* de Windows.

Aunque por el lado de Windows se ha encontrado un desarrollo de las librerías de SCTP, al cual se puede acceder con el siguiente enlace http://www.sctp.be/sctplib/index.htm (Ultima vez visitado 18-07-2007), de donde se puede descargar el ejecutable *sctplib-1.0.4.exe* el cual instala las librerías *glib-1.2.dll, gmodule-1.2.dll y gthread-1.2.dll*, en la carpeta de *Archivos de programas;* además de ello trae dos aplicaciones sencillas para probarlos. Esta hecho para trabajar con Microsoft Windows XP, sin embargo, vanos han sido los intentos que se realizaron por hacer que esta librería funcione, incluso sus creadores recomiendan usar Linux.

La librería de SCTP, sí está implementada para trabajar con Linux, esto comprueba una vez más el poder de flexibilidad del software libre. Sin embargo, también se tuvo que definir la distribución de Linux a utilizar y el IDE o entorno de programación.

Para definir la distribución se probó con las versiones de Mandriva 2007, Ubuntu 7.04 Feisty, y Centos 4, siendo escogida Ubuntu, por su capacidad de reconocimiento de hardware, ya que reconoció las interfaces de red ethernet e inalámbrica sin mucho problema de la PC principal de trabajo, además de su entorno GNOME muy amigable, y sobre todo de la existencia de los repositorios de fácil uso con la herramienta Synaptic, con la cual podemos encontrar la librería de SCTP.
Para descargar dicha librería en Ubuntu solo necesitamos escoger ***sctplib1, sctplib1-dev y sctplib1-doc*** en el Synaptic y se descargarán todas las dependencias necesarias para poder correr toda aplicación programa sobre SCTP. O también podemos hacerlo por el terminal usando el *apt-get.*

```
$sudo apt-get install sctplib1 sctplib1-dev sctplib1-doc
```

Cabe mencionar que la mayoría de distribuciones de Linux ya traen incluidas las librerías de SCTP, sin embargo, se recomienda instalar las últimas actualizaciones de estas librerías para evitar problemas, pues cuando se probó el cliente servidor sencillo, funcionó sin problemas, pero luego de ello al probar el *Iperf, Netperf, SCTPperf*, se experimentaron algunos inconvenientes, los cuales se resolvieron con las instalación de estas librerías.

Si se utiliza otra versión de Linux se recomienda instalar la librería de SCTP, la cual se puede descargar del siguiente enlace http://sourceforge.net/projects/lksctp(Ultima vez visitado 18-07-2007), la última versión es la 1.0.7 la cual salió el 16-07-2007*, lksctp-2.6.22-1.0.7*. Esta librería es compatible con POXIS (*Linux/BSD/UNIX-like Oses*)

Aunque dado que el ambiente de trabajo es el Laboratorio de Redes de la Universidad, en el cual se usa la distribución Centos 4, esta distribución también fue utilizada. Con esta distribución se pudo probar la librería de SCTP descargada desde *sourceforge*. Sin embargo para las pruebas finales, por comodidad se hicieron todas en Ubuntu.

Con la herramienta Synaptic también podemos descargar e instalar muchas aplicaciones como son los IDE[10], y las librerías que estos IDE necesitan para compilar los programas hechos en un lenguaje dado.

Para definir el IDE a utilizar se revisaron algunos de los diferentes IDE's que ofrecen la posibilidad de programar en C/C++ en Linux, estos son:

- Emacs
- Netbeans
- Eclipse
- EasyEclipse
- KDevelop

[10] IDE, Integrated Development Environment ('IDE') es una aplicación compuesta por un conjunto de herramientas necesarias para un programador.

- Anjuta

En Linux se puede compilar las aplicaciones programadas en lenguaje C fácilmente con el compilador *gcc*, sin embargo, si se desea programar bastantes líneas de código es necesario utilizar un IDE para que de esta manera se haga menos trabajoso programar. Después de probar los IDE antes mencionados se recomienda utilizar Anjuta para programar aplicaciones hechas en lenguaje C sobre Ubuntu, ya que es un entorno muy amigable e inductivo, además es compatible y de fácil interacción con Glade.

Glade es una herramienta de desarrollo que ayuda a crear de manera sencilla el código fuente en C de la interfaz gráfica para el usuario, la cual se puede hacer en base a GTK o GNOME.

Emacs es un entorno similar a trabajar con un editor de textos y el *gcc,* por lo que es útil para aplicaciones sencillas, pero cuando se requiere trabajar con una aplicación más compleja, es mejor trabajar con IDE que ofrezca mayores funcionalidades.

Tan Eclipse como Netbeans necesitan de un parche adicional al IDE para poder programar en C/C++, ya que fueron inicialmente diseñados para la programación en Java. Sin embargo ambos son pesados, por lo cual se puede optar por EasyEclipse, el cual es un entorno más ligero que fue diseñado a partir de Eclipse, sin embargo estos tres IDE son ideales para trabajar con Java mas no en C.

Kdevelop es similar a Anjuta, pero Anjuta es mucho más intuitivo.

Luego de ello se paso a investigar cual sería la mejor aplicación para poder comparar los tres protocolos, ya que ese es el principal objetivo de este trabajo. Se encontró que ya existían herramientas que se podía utilizar para el análisis, las cuales se describen a continuación.

3.2 Software existente que ya utiliza SCTP

Para llegar al objetivo principal de este trabajo de Tesis, el cual es comparar el desempeño de los protocolos, utilizamos las siguientes aplicaciones:

3.2.1 Iperf

Iperf es una herramienta muy útil, la cual ha sido diseñada para medir el *perfomance* de una red, permite dar reportes en función a los parámetros de ancho de banda, *throughput*, *jitter*, retardo, y perdida de paquetes, para tráfico TCP y UDP.

Iperf esta corriendo en el puerto 5050 del servidor, y soporta IPv6. Actualmente Iperf se encuentra en su versión 2.0.2 la cual podemos descargar de http://dast.nlanr.net/Projects/Iperf/ (Ultima vez visitado 18-07-2007), y la versión 1.0.7 para Windows.[20]

Las características de Iperf para los protocolos de transporte son:

- Para TCP:
  - Medida del ancho de banda.
  - Reporta el tamaño de MSS/MTU.
  - Soporte para ventana de TCP vía *socket buffers*.
  - Conexiones múltiples simultáneas.
- Para UDP:
  - El cliente puede crear flujos UDP y especificar su tamaño
  - Medida de pérdida de paquetes
  - Soporta *Multicast*
  - Conexiones múltiples simultáneas.

Para mostrar un ejemplo de cómo funciona el Iperf, veamos un ejemplo en el cual se busca el ancho de banda que maneja uno de los servidores locales, para ello usamos el siguiente comando:[20]

iperf -c 192.168.203.143

```
iperf -c 192.168.203.143
In TCP mode
------------------------------------------------------------
Client connecting to 192.168.203.143, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.203.179 port 39209 connected with
192.168.203.143 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   112 MBytes  94.1 Mbits/sec
  close failed: Bad file descriptor
```

La versión Iperf1.6.5 fue modificada para trabajar con la librería *lksctp*, es decir una implementación sobre SCTP, esto se detallará mejor en la etapa de pruebas del siguiente capítulo. Esta versión puede ser descargada del enlace http://datatag.web.cern.ch/datatag/WP3/sctp/iperf-1.6.5-hb-lksctp.tar.gz (Ultima vez visitado 18-07-2007).

3.2.2 Netperf

Netperf es una herramienta que se puede usar para medir varios aspectos de *performance* en una red, con esta herramienta se puede obtener reportes en función a los parámetros de ancho de banda, *throughput*, *jitter*, retardo, y perdida de paquetes, para tráfico TCP y UDP y ahora también para SCTP. Permite hacer pedidos de transferencia unidireccional sobre IPv4 e IPv6, haciendo una interfaz de *sockets* sobre UDP, TCP y SCTP.

Netperf puede trabajar sobre las siguientes plataformas:

- Unix – en la mayoría de sus variaciones.
- Linux
- Windows
- OpenVMS
- Others

Netperf tiene el código abierto, se pueden hacer mejoras, y modificaciones, y se pueden enviar los cambios a la siguiente dirección de correo: netperf-feedback@netperf.org. Netperf es desarrollada y actualizada informalmente por Rick Jones. Sin embargo, se decidió usar otras herramientas ya que netperf aún se encuentra en desarrollo, por ejemplo aún no ofrece la posibilidad de hacer pruebas de *Multistreaming.*[21]

3.2.3 SCTPperf

SCTPperf es una herramienta muy útil, la cual ha sido diseñada para medir el correcto perfomance del protocolo SCTP en una red, permite especificar las opciones para probar el *multi-homing* y el *multi-streaming*. El autor de esta herramienta es Pawel Hadam, y puede ser descargada desde el siguiente link

http://drakkar.imag.fr/users/Pawel.Hadam/work/sctpperf/index.html (Ultima vez visitado 18-07-2007).

Trabaja en forma de cliente y servidor, similar a *iperf* y a *netperf* . SCTPperf en su versión v.0.1 ha sido diseñado para trabajar con un kernel de Linux 2.6.7 o superior, junto con la librería de SCTP lkSCTP 1.0.1. Aunque se puede añadir que con *sctplib1,* que aparece en los repositorios de Ubuntu, trabaja muy bien. Actualmente SCTPperf ha dejado de desarrollarse. Es de software libre, por lo cual se pueden hacer modificaciones para mejorarlo, o bien reparar posibles fallas. SCTPperf permite dar reportes en función a los parámetros de ancho de banda, *throughput*, *jitter*, retardo, y pérdida de paquetes. SCTPperf soporta solamente IPv4.

Una vez descargado debemos de compilarlo y aparecerán dos ejecutables *./scptperf_serv* y *./scptperf_clnt.*

Se divide en dos partes, el servidor tiene los siguientes parámetros:[22]

```
./scptperf_serv

   -P  local_port  → Puerto local
   -H local_host   → Dirección IP local
   -B local_host1  → Dirección IP local adicional
   -f size_kB      → Tamaño del buffer en KB
   -v              →  Verbose mode

Fuente:http://drakkar.imag.fr/users/Pawel.Hadam/work/sctpperf/index.html
```

El cliente tiene los siguientes parámetros:

```
./scptperf_clnt

   -P local_port   → Puerto local
   -H local_host   → Dirección IP local
   -B local_host1  → Dirección IP local adicional
   -h remote_host  →  Dirección IP remota
   -p remote_port  →  Puerto remoto
   -l message_size →  Tamaño de mensaje
   -t time_to_run  →  Tiempo total de ejecución
   -x period       →  Período
   -m streams      →  Número de streams
```

```
   -f size_kB      →  Tamaño del buffer en KB
   -v              →  Verbose mode

Fuente:
http://drakkar.imag.fr/users/Pawel.Hadam/work/sctpperf/index.html
```

Así por ejemplo se puede configurar el servidor para que trabaje con una sola dirección, e igual el cliente con los siguientes comandos.

./scptperf_srv -P 2002 -H 127.0.0.1

./scptperf_clnt -P 2007 -H 127.0.0.1 -p 2002 -h 127.0.0.1 -l 1000 -t 4 -x 1

Y en el caso de usar *muti-homed*

./scptperf_srv -P 2002 -H 127.0.0.1 -B 10.0.0.1 -B 10.1.1.1

3.2.4 EchoTools

EchoTools es una herramienta muy útil, fue creada –como Tesis de Maestría – por el Brasileño Gustavo do Carmo. Echo Tools es una herramienta que ha sido diseñada para comparar el *perfomance* de los protocolo SCTP y TCP.

Al igual que las herramientas anteriores, también trabaja bajo el modelo cliente – servidor.

La versión inicial esta en portugués. Como se ha utilizado esta aplicación para varias pruebas en este trabajo de investigación, se ha realizado algunos cambios, como son el Idioma Español en los mensajes de salida, los nombres de las variables de trabajo, los nombres de las funciones, e incluso los comentarios en cada parte del código fuente. Esto con el objetivo de hacerla más amigable para usuarios Hispano hablantes, y para hacer más fácil el trabajar con la aplicación en la etapa de pruebas.

También se ha trabajando en agregarle una interfaz grafica de usuario con la herramienta Glade y Anjuta, sin embargo, se deja esta tarea como trabajo futuro, en el siguiente paso de esta Tesis.

EchoTools tiene dos partes, el servidor es llamado *EchoServer.c* y el cliente es llamado *GeneraPaquetes.c*

El Servidor:

```
./EchoServer -H dirIPLoc -P ptLoc -t tipProt -b buffer -d orden

-H dirIPLoc → direccion IP  local
-P ptLoc    → puerto local
-t tipProt  → protocolo de transporte - TCP=1 - SCTP=2
-b buffer   → buffer de envío y recepción (Kbytes) min 1 máx. 1000
-d orden    → 1  envía los mensajes de forma desordenada-solo en SCTP
```

El Cliente:

```
./GeneraPaquetes -H dirIPLoc -P ptLoc -h dirIPRem -p ptRem -t tipProt
                 -x numFlujos -f fila -b buffer -m Mensajes -s size
                 -d orden -u tiempo -v verbose

-H dirIPLoc → dirección IP local
-P ptLoc    → puerto local
-h dirIPRem → dirección IP Remota
-p ptRem    → puerto Remoto
-t tipProt  → protocolo de transporte - TCP=1 - SCTP=2
-x numFlujos→ número de flujos - min 1 - máx. 300 - SCTP
-f fila     → fila de Mensajes en cada flujo - min. 1 - máx. 5 - SCTP
-b buffer   → buffer de envió y recepción (kBytes) - min. 1 - máx.
1000 - SCTP
-m Mensajes → número de Mensajes a enviar - min 1 - máx.  100
-s size     → tamaño da mensaje (bytes) - min 2 - máx.  100
-d orden    → 1 - envía los Mensajes de manera desordenada
-u tiempo   → tiempo entre Mensajes - milisegundos - patrón 50
-v verbose  → verbose 1 o 2
```

Con esta herramienta se puede hacer variar los parámetros de número de *streams*-flujos, filas (que son el numero de mensajes que pueden haber en cola en mismo flujo), el número de mensajes. Y de esta forma ver qué protocolo tiene mejor *perfomance*. Y en qué casos conviene utilizar un mayor número de *streams*.[7]

CAPITULO IV ANÁLISIS DE COMPARACIÓN DEL DESEMPEÑO DE SCTP CON TCP Y UDP

Para esta parte vamos a utilizar algunas de las herramientas mencionadas en el anterior capítulo, buscando demostrar la superioridad del protocolo SCTP.

Inicialmente mostraremos las pruebas con el Iperf modificado a TCP-style. Luego mostraremos los análisis y comparación del protocolo TCP y SCTP usando la herramienta EchoTools, donde se mostrará como se comporta el protocolo SCTP trabajando con un solo flujo, y cómo se comporta cuando trabaja con una mayor cantidad de flujos.

Finalmente con la herramienta SCTPperf se mostrará un análisis de la funcionalidad de *multihoming* que tiene SCTP, y como este opera, incluso se mostrará las capturas para ver como el protocolo deja de utilizar una interfaz para usar la otra.

4.1 Pruebas con Iperf

Para esta parte solo se utilizó un escenario, del que constan dos computadoras portátiles y un conmutador, conectados por interfaz *ethernet*.

En el capítulo anterior se mostró la forma de utiliza la herramienta Iperf, la versión *iperf-1.6.5-hb-lksctp* ha sido modificada para trabajar con el protocolo SCTP y se recomienda usar en conjunto con la librería *lksctp-2_5_65-0_6_8* [11], para trabajar con TCP o UDP no hay que hacer ningún cambio, en cambio para trabajar SCTP es necesario agregar " –z " al comando, un ejemplo se muestra en la figura A2.3 perteneciente al Anexo2.

Se realizó veinte experimentos comparando el ancho de banda y la cantidad de datos transmitidos, los cuales se pueden encontrar en la Tabla A2.1 y las gráficas que representan estos resultados son la figura A2.1 y la figura A2.2; estos resultados muestran al protocolo TCP ligeramente superior al protocolo SCTP, ya que supera en ancho de banda obteniéndose hasta 94.1Mbps del enlace de 100Mbps que se tiene con *ethernet*, mientras que solamente 92.3Mbps para SCTP. TCP llega a transmitir 112MB en promedio, mientras que SCTP 109.8MB.

Aparte se encontraron los experimentos hechos por *Asim Iqbal*, los cuales se encuentran en la Web http://datatag.web.cern.ch/datatag/WP3/sctp/tests.htm [19] y también en el Anexo 2, en la tabla A2.2, en la figura A2.3 y en la figura A2.4. Donde se muestra pruebas hechas desde una estación en Geneva hasta otra en Chicago con un enlace de 1Gbps, este experimento también confirma los datos de las pruebas que antes se mencionaron.

Dado que se encontró un desempeño mejor para TCP, se decidió analizar el código fuente de la aplicación Iperf 1.6.5 partiendo de la suposición de que en esta herramienta se hicieron pocas modificaciones para que trabaje como TCP-*Style,* con lo cual no se habrían implementado todas las funcionalidades adicionales que ofrece el protocolo SCTP, ya que la versión 1.6.5 fue modificada en el año 2003. Se asumió que la aplicación solo hacia el cambio de la variable *IPPROTO_TCP* por *IPPROTO_SCTP,* con lo cual tendría el mismo funcionamiento que usando el protocolo TCP, pero enviando mensajes SCTP.

[11] lksctp-2_5_65-0_6_8 se puede descargar desde http://sourceforge.net/project/showfiles.php?group_id=26529

Luego de analizar todo el código de la aplicación, se encontró que efectivamente la suposición era correcta. En el archivo *Socket.cpp* se había hecho la modificación principal y el resto era modificaciones adicionales para que existiese una opción adicional y funciones que en lugar de TCP lleven el nombre SCTP. Veamos el extracto de código:

```
...
int type = ( (  selectedProtocol == UDP )  ?
SOCK_DGRAM : SOCK_STREAM);
...
if (  selectedProtocol != SCTP ) {
          mSock = socket( domain, type, 0 );
           FAIL_errno( mSock == INVALID_SOCKET,
"socket" );
      }
      else if (  selectedProtocol == SCTP) {
          mSock = socket( domain, type, IPPROTO_SCTP );
          FAIL_errno( mSock == INVALID_SOCKET, "socket"
);                  }
...
 //Fuente: Socket.cpp de iperf-1.6.5-hb-lksctp.tar.gz
```

Notemos que en la primera parte se selecciona el tipo de *socket*, sea SOCK_STREAM o SOCK_DGRAM., con un condicional. Luego en la segunda parte notamos que cuando no se escoge SCTP, se pone en 0 el último parámetro de la función *socket()*, y se pone cero por defecto ya que no es necesario poner IPPROTO_TCP, ni IPPROTO_UDP, ya que con el parámetro *type* se identifica el tipo de socket de esta manera se identifica si TCP o UDP. Sin embargo para SCTP si es necesario.

Por lo tanto la herramienta *iperf-1.6.5-hb-lksctp* no tiene lo suficiente para poder realizar la comparación con el protocolo SCTP, ya que no puede mostrar todas sus ventajas.

El hecho de que TCP tenga un menor *performance* que SCTP con TCP-*style,* puede ser atribuido a *overhead,* es decir que SCTP este introduciendo una mayor cantidad de cabeceras.

Ahora si comparamos la cabecera de TCP y SCTP, notaremos que la cabecera de TCP lleva la información de control y luego de ello los datos, en cambio SCTP, aunque tiene una cabecera más pequeña, la información de control como los datos son enviados en *chunks*, luego de la cabecera de SCTP, y cada uno de estos *chunks* tiene su propia cabecera, por lo tanto, SCTP al usar TCP-style (un único *stream*) introduce mayor *overhead* que TCP.

Y es lógico pensar que si un protocolo trabaja de manera similar a TCP, es decir utilizando las mismas llamadas de peticiones y la misma API, lo mejor que pueda hacer es igualar el *performance* de TCP o se puede tener incluso un menor rendimiento, por lo tanto fue necesario probar con otras herramientas.

4.2 Pruebas con EchoTools

En esta parte se utilizaron tres escenarios (Ver figura 4.1):

Escenario 1: Dos computadoras portátiles conectadas a través de un conmutador donde se corre el EchoTools usando una conexión TCP.

Escenario 2: Dos computadoras portátiles conectadas a través de un conmutador donde se corre el EchoTools usando una asociación SCTP donde se usa un solo *stream* para la transmisión. Este escenario es similar al primero. Supuestamente este es el caso equivalente a trabajar con TCP, usando UDP *Style sockets*, por lo que debería ser más eficiente.

Escenario 3: Dos computadoras portátiles conectadas a través de un conmutador donde se corre el EchoTools usando una asociación SCTP donde se usan varios *streams* para la transmisión, por lo que se usa la propiedad de *multi-streaming* de SCTP.

En base a estos tres escenarios se hacen las pruebas en dos fases:

Fase 1: Se varía el tamaño de los mensajes en cada escenario.

Fase 2: Se varía el número de mensajes en cada escenario.

Los datos de los equipos utilizados se encuentran en el Anexo 2, en la figura A2.29.

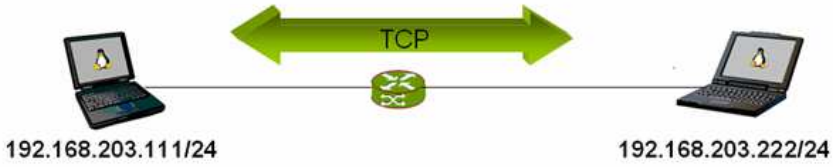
El tipo de mensajes utilizado por el EchoTools es Gnutella, el cual es un protocolo de distribución de archivos en red. Se utilizan los siguientes mensajes:

PING: se usa para conseguir datos de otros nodos en la red.

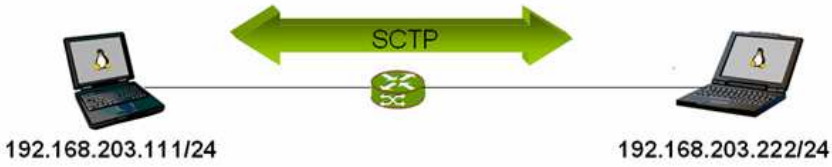
PONG: es la respuesta al mensaje PING, contiene una dirección IP de uno de los participantes que ofrece sus servicios para compartir archivos, el número de archivos.

QUERY: es usado para buscar una cadena de caracteres en el otro extremo, podría ser un archivo en especial, lleva la velocidad a la que el nodo origen puede compartir archivos.

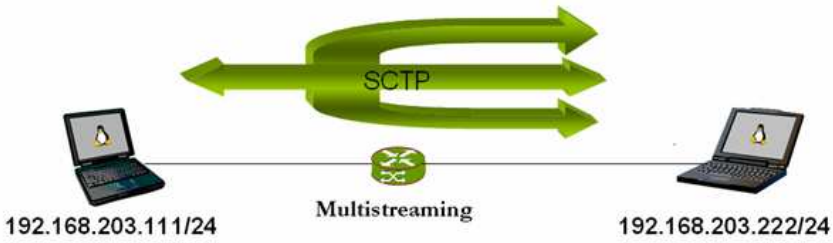
QUERYHIT: es la respuesta al mensaje QUERY, lleva el nombre del archivo, su tamaño y otros datos opcionales.



a) Escenario 1 : Una conexión TCP



b) Escenario 2 : Una asociación SCTP con un stream



c) Escenario 3 : Una asociación SCTP con múltiples streams

Figura4.1. ESCENARIOS DE PRUEBAS DE COMPARACIÓN TCP VS SCTP

4.2.1 Variando el tamaño de los mensajes

Para este caso hicimos lo siguiente:

Para el escenario 1:

- El servidor fue configurado con los siguientes parámetros:

```
Direccion IP del host local:192.168.203.222
Puerto del host local:2222
Protocolo de transporte = TCP
Buffer de envio y recibimiento (Kbytes):100
Entrega de mensajes: ordenada
```

- El Cliente fue configurado con los siguiente parámetros

./GeneraPaquetes -H 192.168.203.111 -P 1111 -h 192.168.203.222 -p 2222 -t 1 -m 50 -s 10

```
DireccionIP del host local:192.168.203.111
Puerto del host local:1111
DireccionIP del host Remoto:192.168.203.222
Puerto del host Remoto:2222
Protocolo de transporte: TCP
Numero de Mensajes a enviar:50
tamano de Mensajes (bytes):10
Intervalo entre Mensajes (milisegundos):0
```

Para el escenario 2:

- El servidor fue configurado con los siguientes parámetros:

```
Direccion IP del host local:192.168.203.222
Puerto del host local:2222
Protocolo de transporte = SCTP
Buffer de envio y recibimiento (Kbytes):100
Entrega de mensajes: ordenada
```

- El Cliente fue configurado con los siguiente parámetros

./GeneraPaquetes -H 192.168.203.111 -P 1111 -h 192.168.203.222 -p 2222 -t 2 -x 1 -f 5 -m 50 -s 10

```
DireccionIP del host local:192.168.203.111
Puerto del host local:1111
DireccionIP del host Remoto:192.168.203.222
Puerto del host Remoto:2222
Protocolo de transporte: STCP
```

```
Numero de flujos:1
Fila de Mensajes en cada flujo:5
Buffer de envio y recepcion configurado como patron
Entrega de Mensajes: ordenada
Numero de Mensajes a enviar:50
tamano de Mensajes (bytes):10
Intervalo entre Mensajes (milisegundos):0
```

Para el escenario 3:

- El servidor se mantiene la configuración anterior, y solo se varía el cliente
- El Cliente mantiene con la configuración anterior, solo se va variando el número de *streams*.

Sobre los resultados

En cada uno de los escenarios anteriores vamos variando el tamaño de los mensajes tal como se muestra en la tabla A2.3 y la tabla A2.4, se muestran de los datos que se pueden medir con el EchoTools:

Latencia Media.- la media aritmética de todas las latencias individuales, en segundos.

Velocidad General.- la suma de todas las latencias, sin considerar los intervalos, en Kbps.

Se muestran seis gráficas de la figura A2.4 a la figura A2.9 y la tabla A2.3 para la latencia media de donde se observa que:

Se corrió el programa una vez con TCP ya que no se puede hablar de *streams,* y varias veces con SCTP para mostrar los diferentes casos al utilizar un diferente y cada vez mayor número de *streams,* así se corrió el programa usando un *stream* hasta usar 300 *streams*, podemos comparar el desempeño de TCP con cada uno de los casos, es decir desde compara TCP versus SCTP 1 *stream,* TCP versus SCTP 10 *streams,* así sucesivamente hasta comparar TCP versus SCTP 300 *streams* (ver figura A2.4), donde se muestra la curva de TCP encima de las curvas de SCTP, lo que implica que TCP demoró más para enviar la misma cantidad de datos.

Comparando el *performance* de TCP versus SCTP con un solo *stream*, SCTP tiene una menor latencia (ver figura A2.5), recordar que este es el caso donde SCTP se comporta como TCP, pero a diferencia del caso del Iperf1.6.5, el EchoTools esta programado en UDP-*Style,* además de ello podemos compararlo también con 10 *stream* y 50 *streams* que son los casos de la figura A2.6 y figura A2.7 respectivamente, donde por intervalos se encuentra una latencia muy baja. Por lo que también se comparó los casos de SCTP para analizar cuantos flujos es mejor.

En la figura A2.8 se encuentran los casos con pocos *stream*s, donde se ve que el caso de 150 *stream*s es el que tiene una menor latencia en relación a usar 1, 10, 50 o 100 *streams*.

En la figura A2.9 se encuentra los casos con muchos *streams*, 150, 200, 250 y 300 *streams*, notemos que se observa una menor latencia y por tanto un mejor *performance* en los casos de 150, 200, 250, en cambio para 300 es similar a tener pocos *streams*, por lo que podemos notar que no es el mejor caso el tener más *streams*, ni tampoco el que tiene un solo *stream*, sino que tenemos que buscar el caso ideal que se ajusta a la aplicación que estemos usando, ya que podríamos estar creando un exceso de flujos o una deficiencia de lo que en realidad se necesita, por ejemplo si necesitamos enviar 100 archivos que ocupan regular espacio en memoria y que necesitamos transmitirlos al mismo tiempo, y usamos 5 *streams,* uno para el control y cuatro para transmitir los archivos; si usamos 30 *streams,* uno para el control y 29 para transmitir los archivos, el mejor caso será el segundo pues se podrá enviar más archivos al mismo tiempo, pero si llegáramos a utilizar 300 *streams*, estaremos sobre dimensionando la cantidad de flujos necesario*.* Para el caso del experimento anterior donde se enviaba 50 mensajes de distintos tamaños, el mejor *performance* se vio al utilizar 150 *stream*s.

Ahora analizando la otra variable, la Velocidad General, se muestran seis gráficas de la figura A2.10 a la figura A2.15 y la tabla A2.4 de donde se observa que:

Nuevamente el *performance* de SCTP es superior a TCP, ya que todas las curvas se encuentran por encima, lo que implica una mayor velocidad, es decir se transmitió una mayor cantidad de datos por unidad de tiempo (ver figura A2.10).

Comparando los casos de TCP y SCTP con pocos *streams*, TCP tiene una menor velocidad general (ver figura A2.11, figura A2.12, figura A2.13), ya que su curva esta por debajo de las demás. Ahora si comparamos solamente las curvas del SCTP (ver figura A2.14 y figura A2.15), notamos que la curva que demuestra un mejor *performance* es para 150 *streams*, similar para el caso en que estábamos analizando la variable de la latencia media. Además se observa que las curvas de 200 y 250 *streams* también tienen un buen *performance*, y la de 300 *streams* similar a trabajar con TCP, lo que demuestra que usar muchos *streams* puede sobrecargar el enlace, en lugar de beneficiarlo, por lo tanto al usar una cantidad de *streams* en demasía se introduce *overhead.*

4.2.2 Variando el número de los mensajes

Para este caso, como antes se menciono, se toman los mismos escenarios, ya que esta es la segunda fase de las pruebas con el EchoTools. No obstante para esta etapa de pruebas se debe de variar el parámetro del número de mensajes. El orden de la toma de datos de las pruebas es el mismo que el anterior.

Para este caso, similar a la primera fase, se analizan también los valores de la "Latencia Media" y la "Velocidad General", ello se encuentra en las tablas Tabla A2.5 y Tabla A2.6 del Anexo 2.

Se muestran seis gráficas de la figura A2.16 a la figura A2.21 para la "Latencia Media" y otras seis de la figura A2.22 a la figura A2.27 para la "Velocidad General" de donde se observa que:

En todos los casos el protocolo TCP presenta una mayor latencia media y una menor velocidad general, cuando es comparada con las curvas del protocolo SCTP para 1 mensaje, 10 mensaje y 50 mensaje. Es decir SCTP tiene un mejor *performance*.

Comparando únicamente las gráficas de SCTP notamos que la gráfica para 50 mensajes es la que mejor *performance* presenta, ya que es la curva con menor latencia media y la curva con mejor crecimiento en cuanto a la velocidad general. Además el hecho de aumentar el número de mensajes no implica que el usar

muchos mensaje sea la solución, ya que los casos de 10, 50, 100, 150 mensajes, presentan mejor velocidad general que los casos donde se usaron más mensajes.

4.3 Pruebas con SCTPperf

En esta parte se añade al escenario anterior un AP (*Access Point*) para agregar un camino adicional. Para ello empezamos a usar las interfaces inalámbricas que poseen ambos, por coincidencia ambas computadoras portátiles tenían tarjetas con *chipset* broadcom4318. La forma de activar estas interfaces la pueden encontrar en el anexo 3.

El modelo del AP utilizado es: Dlink Air Plus G+ 2.4GHz, Wireless Access Point.

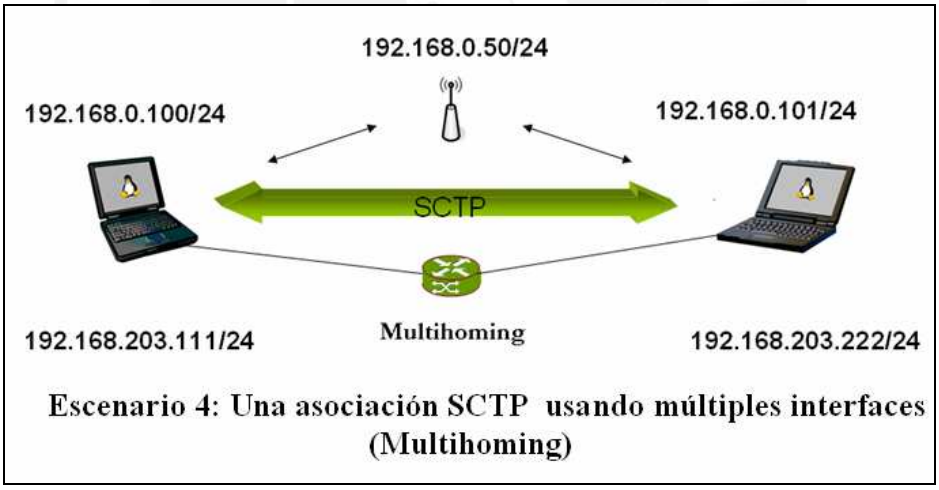


Figura4.2. ESCENARIO DE PRUEBA DE MULTIHOMING

Para esta experiencia primero se ejecuto el sctpperf en cada interfaz por separado, es decir se corrió el sctpperf por la red 192.168.203.0/24 y luego se corrió el sctpperf sobre la red 192.168.0.0/24. Para finalmente utilizar ambas interfaces, y desconectar una, para ver como la aplicación seguía corriendo.

Los resultados de estas pruebas se encuentran en el anexo 2.

4.3.1 SCTPperf sobre ethernet

- El servidor fue configurado con los siguientes parámetros:

./scptperf_srv -P 2222 -H 192.168.203.222 -f 100

```
Local host:                192.168.203.222
Local port:                2222
Sender buffer:            102400
Receiver buffer:          102400
Verbose:                  0

Binding single address → solo una interfaz
```

- El Cliente fue configurado con los siguiente parámetros

./scptperf_clnt -P 1111 -H 192.168.203.111 -p 2222 -h 192.168.203.222 -l 1000 -t 5 -x 1 -m 50 -f 100

```
Local host:                192.168.203.111
Local port:                1111
Remote host:               192.168.203.222
Remote port:               2222
Packet size:              1000
Measure period:           5
Print period:             1
Streams:                  50
Sender buffer:            102400
Receiver buffer:          102400
Verbose:                  0
```

El resultado de esta medida se encuentra en la figura A2.30, se muestra la salida en el cliente, ya que en el servidor se muestran casi los mismos resultados. En resumen el resultado es:

BANDWIDTH = 87.652817 Mbits/s, SIZE = 1000 bytes, SENT = 57449 packets

4.3.2 SCTPperf sobre wireless (802.11b)

- El servidor fue configurado con los siguientes parámetros:

./scptperf_srv -P 2222 -H 192.168.0.101 -f 100

```
Local host:                 192.168.0.101
Local port:                 2222
Sender buffer:              102400
Receiver buffer:            102400
Verbose:                    0

Binding single address → solo una interfaz
```

- El Cliente fue configurado con los siguiente parámetros

./scptperf_clnt -P 1111 -H 192.168.0.100 -p 2222 -h 192.168.0.101 -l 1000 -t 5 -x 1 -m 50 -f 100

```
Local host:                 192.168.0.100
Local port:                 1111
Remote host:                192.168.0.101
Remote port:                2222
Packet size:                1000
Measure period:             5
Print period:               1
Streams:                    50
Sender buffer:              102400
Receiver buffer:            102400
Verbose:                    0
```

El resultado de esta medida se encuentra en la figura A2.31 se muestra la salida en el cliente, ya que en el servidor se muestran casi los mismos resultados. Sin embargo, en resumen el resultado es:

BANDWIDTH = 0.630010 Mbits/s, SIZE = 1000 bytes, SENT = 414 packets

Notamos que 0.63Mbps es un valor muy bajo para el estándar 802.11b, el cual trabaja a 11Mbps, y que este valor debió salir alrededor de 4Mbps. Pero con esta experiencia apuntamos a mostrar la funcionalidad *Multihoming* de SCTP, por lo que no se entra a detallar el porqué de este valor.

4.3.3 SCTPperf sobre ethernet y *wireless* (802.11b) - *Multihoming*

Para este caso, se conectó ambas interfaces y se ejecutó la aplicación para luego retirar la interfaz de red para ver como respondía la aplicación.

Debemos de recordar que SCTP cuando trabaja en *multi-homing* selecciona un camino como primario, y el otro es monitoreado con mensajes *heartbeat*, si detecta

que la interfaz principal se cae entonces revisa si la otra esta disponible y la utiliza como principal.

Para esto también se hicieron capturas con el *Wireshark*[12]

- El servidor fue configurado con los siguientes parámetros:

./scptperf_srv -P 2222 -H 192.168.203.222 -B 192.168.0.101 -f 100

```
Local host:               192.168.203.222
Local port:               2222
Sender buffer:            102400
Receiver buffer:          102400
Verbose:                  0

Binding single address
Binding mutliple addresses from addr_buf[] →  usa dos interfaces
```

Notar que la segunda dirección IP usada no es mostrada por la aplicación, sin embargo, la última línea "`Binding mutliple addresses from addr_buf[]`" muestra que se están usando dos interfaces.

- El Cliente fue configurado con los siguiente parámetros

./scptperf_clnt -P 1111 -H 192.168.203.111 -B 192.168.0.100 -p 2222 -h 192.168.203.222 -l 1000 -t 5 -x 1 -m 50 -f 100

```
Local host:               192.168.203.111
Local port:               1111
Remote host:              192.168.203.222
Remote port:              2222
Packet size:              1000
Measure period:           5
Print period:             1
Streams:                  50
Sender buffer:            102400
Receiver buffer:          102400
Verbose:                  0

Binding single address
Binding mutliple addresses from addr_buf[] →  usa dos interfaces
```

[12] *Wireshark* antes conocido como ethereal, permite capturar diferentes tipos de paquetes.

Ahora analicemos las tramas capturadas, en la figura 4.3 se muestra los trozos más importantes de la captura realizada. Se ha separado en cuatro partes. La parte (a) muestra la inicialización de la asociación la cual es realizada por la interfaz ethernet, por ello las interfaces que aparecen son las de la red 192.168.203.0/24. La parte (b) muestra el cambio de interfaz cuando se desconecta adrede la interfaz ethernet, entonces la aplicación utiliza el segundo camino, y no se detiene. Notemos que pasa de enviar mensajes entre la dirección IP 192.168.203.111 y la dirección IP 192.168.203.222 a enviar mensajes entre la dirección IP 192.168.0.100 y la dirección IP 192.168.0.101.

En la parte (c) se muestran algunos paquetes *heartbeat* que confirman que la interfaz con dirección IP 192.168.0.100 sigue activa. Finalmente en la parte (d) se realiza el término de la asociación y todo por parte de las interfaces inalámbricas.

El cambio de interfaces es transparente para la aplicación ya que continúa haciendo tu trabajo de medir el *throughput*. El resultado de esta medida se encuentra en la figura A2.32 donde se muestra la salida en el servidor, ya que en el cliente se muestran casi los mismos resultados. En resumen el resultado es:

BANDWIDTH = 5.300504 Mbits/s, SIZE = 100000 bytes, RECEIVED = 1568 packets

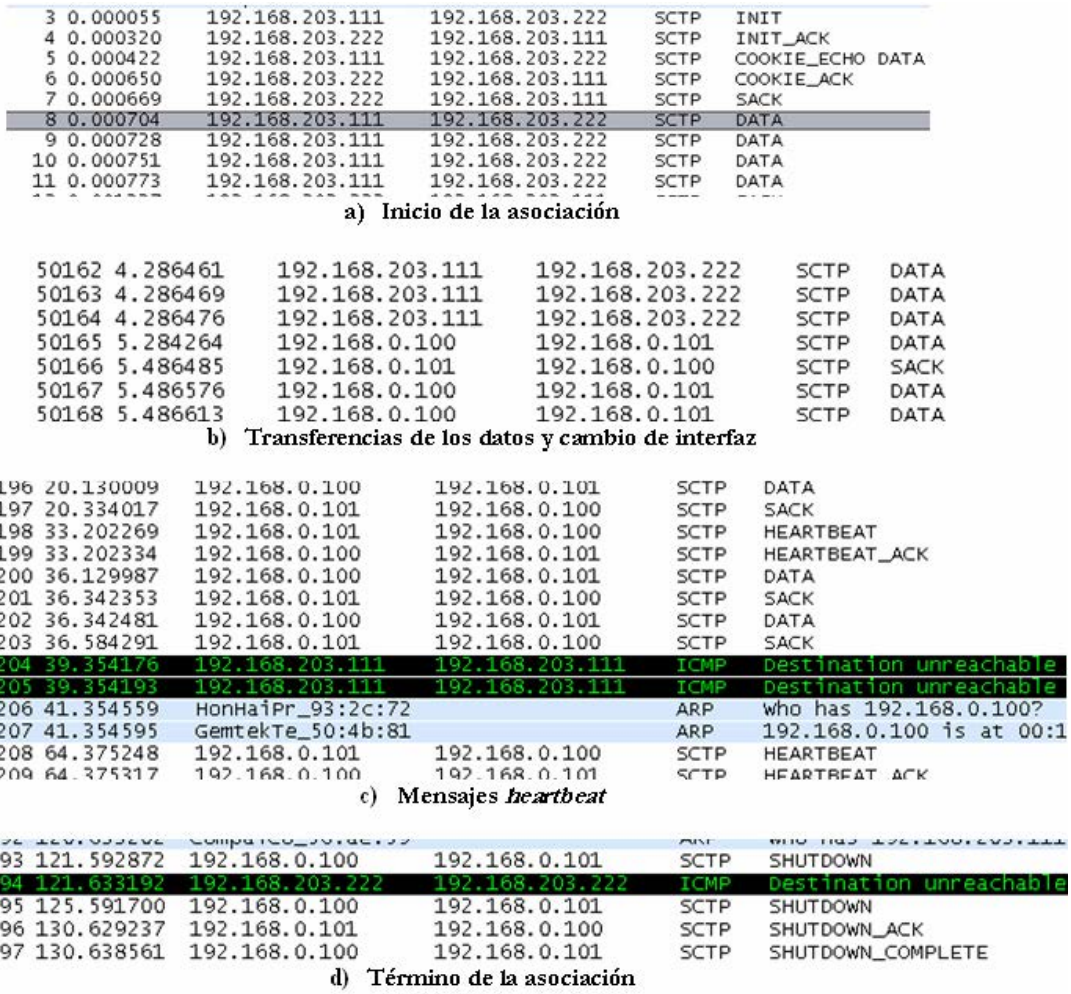


Figura4.3. CAPTURA DE TRAMAS EN EL ESCENARIO MULTIHOMING

4.3.4 SCTPperf *Multihoming* con varias interfaces de red

En esta prueba se partió de la hipótesis que si SCTP maneja *multihoming* al usar una mayor cantidad de interfaces se debe tener un mejor *performance*, por lo que utilizamos el SCTPperf para este escenario de la figura 4.4.

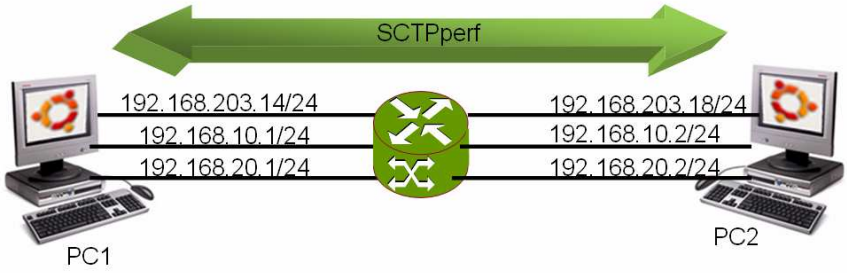
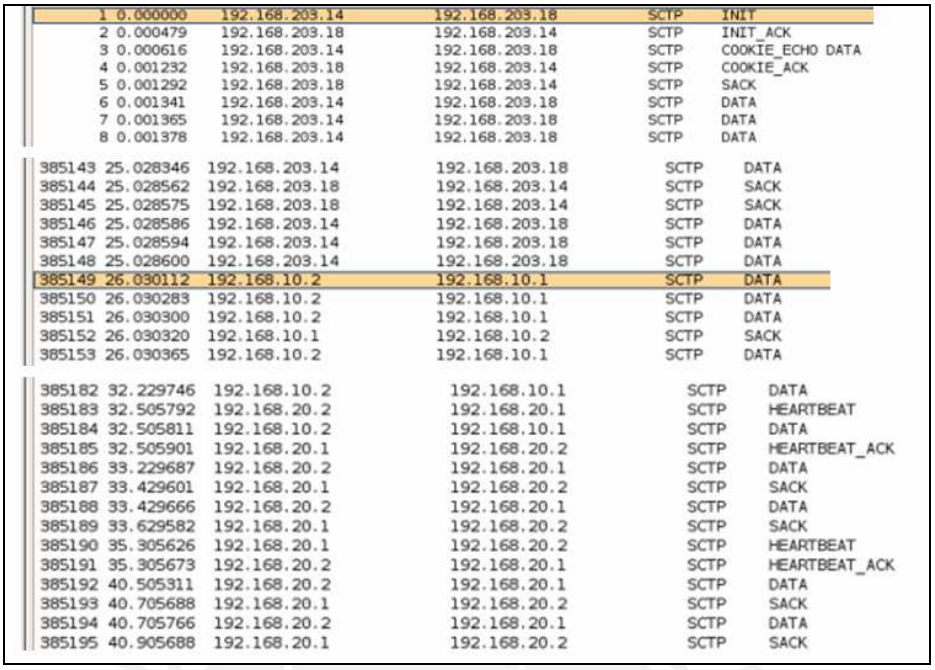


Figura4.4. ESCENARIO MULTIHOMING CON TRES INTERFACES DE RED

Luego de correr la aplicación, notamos que no esta programada para trabajar con las dos o más interfaces en conjunto, sino que utiliza una como principal y las demás como respaldo. Ya que la transmisión solo se daba por uno de los tres enlaces.

Probamos desconectado la interfaces principal de la red 192.168.203.0/24, y notamos que luego empieza a utilizar la interfaz de la red 192.168.10.0/24, desconectamos este enlace y comenzó a usar el último enlace 192.168.20.0/24, esta parte es transparente para el usuario al igual que el caso anterior. En la figura 4.5 se muestra la captura de los resultados, donde podemos apreciar, que la aplicación pasa de una red a otra y no usa las tres interfaces al mismo tiempo, y solo usa uno u otra cuando encuentra problemas en la interfaz que escogió como principal, y se queda monitoreando las demás interfaces con mensajes *HEARTBEAT.*



1	0.000000	192.168.203.14	192.168.203.18	SCTP	INIT
2	0.000479	192.168.203.18	192.168.203.14	SCTP	INIT_ACK
3	0.000616	192.168.203.14	192.168.203.18	SCTP	COOKIE_ECHO DATA
4	0.001232	192.168.203.18	192.168.203.14	SCTP	COOKIE_ACK
5	0.001292	192.168.203.18	192.168.203.14	SCTP	SACK
6	0.001341	192.168.203.14	192.168.203.18	SCTP	DATA
7	0.001365	192.168.203.14	192.168.203.18	SCTP	DATA
8	0.001378	192.168.203.14	192.168.203.18	SCTP	DATA
385143	25.028346	192.168.203.14	192.168.203.18	SCTP	DATA
385144	25.028562	192.168.203.18	192.168.203.14	SCTP	SACK
385145	25.028575	192.168.203.18	192.168.203.14	SCTP	SACK
385146	25.028586	192.168.203.14	192.168.203.18	SCTP	DATA
385147	25.028594	192.168.203.14	192.168.203.18	SCTP	DATA
385148	25.028600	192.168.203.14	192.168.203.18	SCTP	DATA
385149	26.030112	192.168.10.2	192.168.10.1	SCTP	DATA
385150	26.030283	192.168.10.2	192.168.10.1	SCTP	DATA
385151	26.030300	192.168.10.2	192.168.10.1	SCTP	DATA
385152	26.030320	192.168.10.1	192.168.10.2	SCTP	SACK
385153	26.030365	192.168.10.2	192.168.10.1	SCTP	DATA
385182	32.229746	192.168.10.2	192.168.10.1	SCTP	DATA
385183	32.505792	192.168.20.2	192.168.20.1	SCTP	HEARTBEAT
385184	32.505811	192.168.10.2	192.168.10.1	SCTP	DATA
385185	32.505901	192.168.20.1	192.168.20.2	SCTP	HEARTBEAT_ACK
385186	33.229687	192.168.20.2	192.168.20.1	SCTP	DATA
385187	33.429601	192.168.20.1	192.168.20.2	SCTP	SACK
385188	33.429666	192.168.20.2	192.168.20.1	SCTP	DATA
385189	33.629582	192.168.20.1	192.168.20.2	SCTP	SACK
385190	35.305626	192.168.20.1	192.168.20.2	SCTP	HEARTBEAT
385191	35.305673	192.168.20.2	192.168.20.1	SCTP	HEARTBEAT_ACK
385192	40.505311	192.168.20.2	192.168.20.1	SCTP	DATA
385193	40.705688	192.168.20.1	192.168.20.2	SCTP	SACK
385194	40.705766	192.168.20.2	192.168.20.1	SCTP	DATA
385195	40.905688	192.168.20.1	192.168.20.2	SCTP	SACK

Figura4.5. CAPTURA ESCENARIO MULTIHOMING CON TRES INTERFACES DE RED

C O N C L U S I O N E S

- El protocolo SCTP ofrece la posibilidad de *multi-streaming,* sin embargo, se debe tener cuidado en la selección de la cantidad de *streams* a usar, ya que, por las pruebas vistas en el capítulo IV, mientras se envíen una mayor cantidad de mensajes en un tiempo dado, será mejor usar una mayor cantidad de flujos. Pero no en exceso, pues se puede introducir *overhead.*

- SCTP ofrece un mecanismo de ajuste automático de número de *streams,* esto se hace al inicio de la asociación, cuando se intercambian los mensajes INIT e INIT-ACK, donde ambos extremos negocian de acuerdo al número máximo de *streams* que ofrece cada extremo, trabajando al final con el menor valor de ambos.

- Se ha mostrado herramientas que comparan el *performance* entre SCTP y TCP, y no así *versus* UDP ya que no ofrece confiabilidad, y por tanto mayor tiene mayor *throughput.* En UDP la tarea de control suele ser delegada a la capa de aplicación. Por ello la comparación con UDP solo fue cualitativamente

- No es necesario comparar TCP-*style versus* UDP-*style,* puesto que este último ofrece todas las funcionalidades que ofrece el protocolo SCTP, por lo tanto trabajar con UDP-*style* es mejor cualitativa y cuantitativamente.

- Debemos de notar que SCTP ofrece una facilidad de migración de TCP a SCTP, sólo es necesario cambiar unos pocos parámetros en el código fuente, y ya se puede usar SCTP con TCP-*style.* Si queremos hacer uso de todas sus funcionalidades, entonces si tendremos que hacer mayores cambios en el código.

- SCTP tiene la capacidad de recuperación ante fallas, ya que cuando se utiliza *Multihoming*, si se cae una de las interfaces, la otra se activa automáticamente, siendo este proceso transparente para la capa de aplicación.

- SCTP con TCP-*style* tiene un desempeño inferior a TCP cuando se utiliza un único *stream,* pues introduce una mayor cantidad de *overhead*.

O B S E R V A C I O N

El protocolo SCTP es un protocolo que necesita seguir desarrollándose, la extensión del API de *sockets* para SCTP aún se encuentra en *draft*, por lo que se siguen presentando cambios. Un ejemplo de ello es el cambio de nombres de *TCP-Style socket* a *one-to-one socket* y de *UDP-Style socket a one-to-many socket*.

RECOMENDACIONES PARA TRABAJOS FUTUROS

- El año 2006 salió el protocolo *Datagram Congestion Control Protocol* (DCCP), Protocolo de Control de Congestión de Datagramas, DCCP es un nuevo protocolo de nivel de transporte orientado a los mensajes, se recomienda hacer un estudio de comparación con el protocolo SCTP.

- Realizar una aplicación similar al Iperf, pudiendo ser esta el EchoTools integrando en ella herramientas para comparar el *performance* entre TCP, SCTP y DCCP.

- Hacer un estudio sobre los algoritmos de control de congestión, y buscar cual o cuales serían los óptimos para trabajar con SCTP.

- Agregar una interfaz gráfica de usuario al EchoTools para que sea similar al Jperf.

- Actualizar el parche de SCTP del simulador de Qualnet 3.9 a la versión actual.

- Hacer un estudio más profundo sobre los avances de SCTP sobre Windows y sobre las librerías que están en desarrollo para Java.

- Actualizar el servidor httpd y mozilla los parches para que funcionen con SCTP, y agregarlos a los repositorios de Ubuntu.

- Modificar la última versión del VLC para que trabaje con SCTP haciendo uso de su configuración de confiabilidad parcial.

- Realizar un estudio de la aplicación del protocolo SCTP en conjunto con el protocolo IPv6 a las redes celulares, SCTP/IPv6 *mobil* en lugar de TCP/IP.

B I B L I O G R A F Í A

1. Cisco System, Programa de la Academia de Networking de Cisco, Cisco System 2003)

2. Stevens R., UNIX Network Programming Volume 1, The Sockets Networking API, Addison Wesley, Third Edition, 2003.

3. Ríos G., Redes de las Computadoras, PUCP, 2005

4. The International Engineering Consortium, Stream Control Transmission Protocol: http://www.iec.org/online/tutorials/sctp/index.html

5. Ding J.W., Stream Control Transmission Protocol, The Management of Information system, Taiwan: http://fs.mis.kuas.edu.tw/~jwding/Internet%20Technology/PPT/Chap-13.ppt

6. Jones T., Better networking with SCTP, 2006: http://www.ibm.com/developerworks/linux/library/l-sctp/

7. Do Carmo G., Peer-to-peer com a utilização do SCTP para aplicativos de Compartilhamento de arquivos, Master Thesis, 2006

8. Iyengar J., Stream Control Transmission Protocol (SCTP), Protocol Engineering Lab, Computer & Information Sciences, University of Delaware

9. Borisov N, Introduction to Unix Network Programming, 2006: http://www.cs.uiuc.edu/class/fa06/cs438/slides/CS438-02.Sockets.pdf

10. Rodriguez A.,The design of a new reliable transport protocol for IP networks, Master Thesis, 2002

11. Information Sciences Institute, RFC793 - Transmission Control Protocol,

12. University of Southern California, September 1981: http://www.ietf.org/rfc/rfc793.txt

13. Postel J., RFC768 - User Datagram Protocol, October 1980: http://www.ietf.org/rfc/rfc768.txt.

14. Stewart R. and Xie Q., Stream Control Transmission Protocol(SCTP): A Reference Guide, Addison Wesley, 2002

15. Ding J.W., Socket Programming, The Management of Information system, Taiwan: http://fs.mis.kuas.edu.tw/~jwding/Internet%20Technology/PPT/Ch16_Socket%20%20Programming.ppt

16. Coene,L. RFC3257-Stream Control Transmission Protocol Applicability Statement, 2002. http://www.sctp.org

17. XIE, Q.; STEWART, R.; SHARP, C. et al. SCTP Unreliable Data Mode Extension(draft-ietf-tsvwg-usctp-00.txt). IETF Network WorkingGroup,2001

18. STEWART,R.;XIE,Q.;YARROLL,L.etal. Sockets API Extensions for StreamControl Transmission Protocol – SCTP. IETF Network WorkingGroup,2004

19. Asim Iqbal, SCTP Performance Tests, 2003: http://datatag.web.cern.ch/datatag/WP3/sctp/tests.htm

20. Iperf: http://dast.nlanr.net/Projects/Iperf/

21. Netperf, Netperf Manual : http://www.netperf.org/netperf/

22. Pawel Hadam, SCTPperf v.0.1: http://drakkar.imag.fr/users/Pawel.Hadam/work/sctpperf/index.html

23. Newmarch, J. , Stream Control Transmission Protocol Associations, Linux Journal, October 2007.

A N E X O S

En este CD se encuentran el Anexo 1, Anexo 2 y Anexo 3.

Anexo 1: Contiene el código fuente utilizado

Anexo 2: Contiene los resultados de las pruebas del Capítulo IV.

Anexo 3: Contiene información útil de software libre en WiFi.

Anexo 1

Código fuente utilizado

Cliente de Flujos 2
Servidor de Flujos 3
streamcount_echoclient.c 5
streamcount_echoserver.c 6
streamsend_echoclient.c 8
streamsend_echoserver.c 12
GeneraPaquetes.c 16
EchoServer.c 37

Cliente de Flujos

tomado de http://es.tldp.org/Tutoriales/PROG-SOCKETS/prog-sockets/x459.html

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
/* netbd.h es necesitada por la estructura hostent ;-) */

#define PORT 3550
/* El Puerto Abierto del nodo remoto */

#define MAXDATASIZE 100
/* El número máximo de datos en bytes */

int main(int argc, char *argv[])
{
   int fd, numbytes;
   /* ficheros descriptores */

   char buf[MAXDATASIZE];
   /* en donde es almacenará el texto recibido */

   struct hostent *he;
   /* estructura que recibirá información sobre el nodo remoto */

   struct sockaddr_in server;
   /* información sobre la dirección del servidor */

   if (argc !=2) {
      /* esto es porque nuestro programa sólo necesitará un
      argumento, (la IP) */
      printf("Uso: %s <Dirección IP>\n",argv[0]);
      exit(-1);
   }

   if ((he=gethostbyname(argv[1]))==NULL){
      /* llamada a gethostbyname() */
      printf("gethostbyname() error\n");
      exit(-1);
   }

   if ((fd=socket(AF_INET, SOCK_STREAM, 0))==-1){
      /* llamada a socket() */
      printf("socket() error\n");
      exit(-1);d
   }

   server.sin_family = AF_INET;
   server.sin_port = htons(PORT);
   /* htons() es necesaria nuevamente ;-o */
   server.sin_addr = *((struct in_addr *)he->h_addr);
   /*he->h_addr pasa la información de ``*he'' a "h_addr" */
   bzero(&(server.sin_zero),8);
```

```
   if(connect(fd, (struct sockaddr *)&server,
      sizeof(struct sockaddr))==-1){
      /* llamada a connect() */
      printf("connect() error\n");
      exit(-1);
   }

   if ((numbytes=recv(fd,buf,MAXDATASIZE,0)) == -1){
      /* llamada a recv() */
      printf("Error en recv() \n");
      exit(-1);
   }

   buf[numbytes]='\0';

   printf("Mensaje del Servidor: %s\n",buf);
   /* muestra el mensaje de bienvenida del servidor =) */

   close(fd);   /* cerramos fd =) */

}
```

Servidor de Flujos

http://es.tldp.org/Tutoriales/PROG-SOCKETS/prog-sockets/x453.html

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
/* netbd.h es necesitada por la estructura hostent ;-) */

#define PORT 3550
/* El Puerto Abierto del nodo remoto */

#define MAXDATASIZE 100
/* El número máximo de datos en bytes */

int main(int argc, char *argv[])
{
   int fd, numbytes;
   /* ficheros descriptores */

   char buf[MAXDATASIZE];
   /* en donde es almacenará el texto recibido */

   struct hostent *he;
   /* estructura que recibirá información sobre el nodo remoto */

   struct sockaddr_in server;
   /* información sobre la dirección del servidor */

   if (argc !=2) {
      /* esto es porque nuestro programa sólo necesitará un
      argumento, (la IP) */
      printf("Uso: %s <Dirección IP>\n",argv[0]);
      exit(-1);
   }
```

```
    if ((he=gethostbyname(argv[1]))==NULL){
       /* llamada a gethostbyname() */
       printf("gethostbyname() error\n");
       exit(-1);
    }

    if ((fd=socket(AF_INET, SOCK_STREAM, 0))==-1){
       /* llamada a socket() */
       printf("socket() error\n");
       exit(-1);d
    }

    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);
    /* htons() es necesaria nuevamente ;-o */
    server.sin_addr = *((struct in_addr *)he->h_addr);
    /*he->h_addr pasa la información de ``*he'' a "h_addr" */
    bzero(&(server.sin_zero),8);

    if(connect(fd, (struct sockaddr *)&server,
       sizeof(struct sockaddr))==-1){
       /* llamada a connect() */
       printf("connect() error\n");
       exit(-1);
    }

    if ((numbytes=recv(fd,buf,MAXDATASIZE,0)) == -1){
       /* llamada a recv() */
       printf("Error en recv() \n");
       exit(-1);
    }

    buf[numbytes]='\0';

    printf("Mensaje del Servidor: %s\n",buf);
    /* muestra el mensaje de bienvenida del servidor =) */

    close(fd);   /* cerramos fd =) */

}
```

streamcount_echoclient.c

tomado de Newmarch Jan, Linux Journal October 2007

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/sctp.h>

#define ECHO_PORT 2013

char *usage_msg = "usage: streamcount_echo_client ip-addr istreams
ostreams";
char *msg = "hello";
void usage () {
      fprintf(stderr, "%s\n", usage_msg);
      exit(1);
}

int main(int argc, char *argv[]){
      int sockfd;
      int len;
      struct sockaddr_in serv_addr;
      int port = ECHO_PORT;
      struct sctp_initmsg initmsg;
      struct sctp_status status;

      if (argc != 4) usage ();

      /* create endpoint */

      sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_SCTP);

      if (sockfd <0) {
            perror("socket creation");
            exit (2);
      }

      /* connect to server */

      serv_addr.sin_family = AF_INET;
      serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
      serv_addr.sin_port = htons(port);

      memset (&initmsg, 0 ,sizeof (initmsg));
      initmsg.sinit_max_instreams = atoi (argv[2]);
      initmsg.sinit_num_ostreams = atoi (argv[3]);
      printf("Asking for: input streams: %d, output streams: %d\n",
initmsg.sinit_max_instreams,
      initmsg.sinit_num_ostreams);

      if (setsockopt (sockfd, IPPROTO_SCTP, SCTP_INITMSG, &initmsg,
sizeof (initmsg))) {
            perror("set sock opt\n");
      }
```

```
      if (connect(sockfd, (struct sockaddr *) &serv_addr,
      sizeof(serv_addr)) < 0) {
            perror("connectx");
            exit(3);
      }
      len = sizeof(status);
      memset(&status, 0, len);

      if (getsockopt(sockfd, IPPROTO_SCTP, SCTP_STATUS, &status, &len)
== -1) {
            perror("get sock opt");
      }
      printf("Got: input streams: %d, output streams: %d\n",
      status.sstat_instrms,   status.sstat_outstrms);

      /* give the server time to do something */
      sleep(2);
      /* no reads/writes are done */
      close(sockfd);
      exit(0);
}
```

streamcount_echoserver.c

tomado de Newmarch Jan, Linux Journal October 2007

```
 #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/sctp.h>
#define ECHO_PORT 2013

char *usage_msg = "usage: streamcount_echo_server istreams ostreams";
void usage() {
      fprintf(stderr, "%s\n", usage_msg);
      exit(1);
}

int main(int argc, char *argv[]) {
      int sockfd, client_sockfd;
      int len;
      struct sockaddr_in serv_addr, client_addr;
      int port = ECHO_PORT;
      struct sctp_initmsg initmsg;
      struct sctp_status status;

      if (argc != 3) usage();

      /* CREATE ENDPOINT */

      sockfd = socket(AF_INET, SOCK_STREAM,IPPROTO_SCTP);

      if (sockfd < 0) {
            perror("socket");
            exit(2);
      }
```

```
      serv_addr.sin_family = AF_INET;
      serv_addr.sin_addr.s_addr = INADDR_ANY;
      serv_addr.sin_port = htons(port) ;

      if (bind(sockfd, (struct sockaddr *) &serv_addr,
sizeof(serv_addr)) == -1) {
            perror("sctp bind");
            exit(2);
      }

      memset(&initmsg, 0, sizeof(initmsg));
      initmsg.sinit_max_instreams = atoi (argv[1]);
      initmsg.sinit_num_ostreams = atoi (argv[2]);
      printf("Asking for: input streams: %d, output streams: %d\n",
initmsg.sinit_max_instreams, initmsg.sinit_num_ostreams);

      if (setsockopt (sockfd, IPPROTO_SCTP, SCTP_INITMSG, &initmsg,
sizeof(initmsg))) {
            perror("set sock op/n");
      }

      /*specify queue */

      listen(sockfd, 5);

      for (;;) {
            len = sizeof(client_addr);
            client_sockfd = accept(sockfd, (struct sockaddr *)
      &client_addr, & len);

            if (client_sockfd == -1) {
                  perror(NULL) ; continue;
            }

            memset(&status, 0, sizeof(status));
            len = sizeof(status);

            if (getsockopt(client_sockfd, IPPROTO_SCTP, SCTP_STATUS,
&status, &len) == -1){
                  perror("get sock opt");
            }

            printf("Got: input streams: %d, ouput streams: %d\n",
status.sstat_instrms, status.sstat_outstrms);

            /* give the client time to do something */

            sleep(2);
            close(client_sockfd);
      }
}
```

streamsend_echoclient.c

tomado de Newmarch Jan, Linux Journal October 2007

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <netinet/sctp.h>
/*include extras*/

#include <errno.h>
#include <string.h>


/*fin include extras*/

#define SIZE 1024
char buf [SIZE];
#define ECHO_PORT 2013
char *usage_msg = "Usar: ./app direccionIP numero_streams_input
numero_streams_output stream\n";

void usage() {
      fprintf(stderr, "%s\n", usage_msg);
      exit(1);
}
/*funciones*/
int
sctp_connectx(int fd, struct sockaddr *addrs, int addrcnt)
{
      void *addrbuf;
      struct sockaddr *sa_addr;
      socklen_t addrs_size = 0;
      int i;

      addrbuf = addrs;
      for (i = 0; i < addrcnt; i++) {
            sa_addr = (struct sockaddr *)addrbuf;
            switch (sa_addr->sa_family) {
            case AF_INET:
                  addrs_size += sizeof(struct sockaddr_in);
                  addrbuf += sizeof(struct sockaddr_in);
                  break;
            case AF_INET6:
                  addrs_size += sizeof(struct sockaddr_in6);
                  addrbuf += sizeof(struct sockaddr_in6);
                  break;
            default:
                  errno = EINVAL;
                  return -1;
            }
      }

      return setsockopt(fd, SOL_SCTP, SCTP_SOCKOPT_CONNECTX, addrs,
addrs_size);
```

```
}

/* This library function assists the user with the advanced features
 * of SCTP.  This is a new SCTP API described in the section 8.7 of
the
 * Sockets API Extensions for SCTP. This is implemented using the
 * sendmsg() interface.
 */
int
sctp_sendmsg(int s, const void *msg, size_t len, struct sockaddr *to,
           socklen_t tolen, uint32_t ppid, uint32_t flags,
           uint16_t stream_no, uint32_t timetolive, uint32_t context)
{
      struct msghdr outmsg;
      struct iovec iov;
      char outcmsg[CMSG_SPACE(sizeof(struct sctp_sndrcvinfo))];
      struct cmsghdr *cmsg;
      struct sctp_sndrcvinfo *sinfo;

      outmsg.msg_name = to;
      outmsg.msg_namelen = tolen;
      outmsg.msg_iov = &iov;
      iov.iov_base = (void *)msg;
      iov.iov_len = len;
      outmsg.msg_iovlen = 1;

      outmsg.msg_control = outcmsg;
      outmsg.msg_controllen = sizeof(outcmsg);
      outmsg.msg_flags = 0;

      cmsg = CMSG_FIRSTHDR(&outmsg);
      cmsg->cmsg_level = IPPROTO_SCTP;
      cmsg->cmsg_type = SCTP_SNDRCV;
      cmsg->cmsg_len = CMSG_LEN(sizeof(struct sctp_sndrcvinfo));

      outmsg.msg_controllen = cmsg->cmsg_len;
      sinfo = (struct sctp_sndrcvinfo *)CMSG_DATA(cmsg);
      memset(sinfo, 0, sizeof(struct sctp_sndrcvinfo));
      sinfo->sinfo_ppid = ppid;
      sinfo->sinfo_flags = flags;
      sinfo->sinfo_stream = stream_no;
      sinfo->sinfo_timetolive = timetolive;
      sinfo->sinfo_context = context;

      return sendmsg(s, &outmsg, 0);
}

/* This library function assist the user with sending a message
without
 * dealing directly with the CMSG header.
 */
int
sctp_send(int s, const void *msg, size_t len,
          const struct sctp_sndrcvinfo *sinfo, int flags)
{
      struct msghdr outmsg;
      struct iovec iov;

      outmsg.msg_name = NULL;
      outmsg.msg_namelen = 0;
      outmsg.msg_iov = &iov;
```

```
	iov.iov_base = (void *)msg;
	iov.iov_len = len;
	outmsg.msg_iovlen = 1;
	outmsg.msg_controllen = 0;

	if (sinfo) {
		char outcmsg[CMSG_SPACE(sizeof(struct sctp_sndrcvinfo))];
		struct cmsghdr *cmsg;

		outmsg.msg_control = outcmsg;
		outmsg.msg_controllen = sizeof(outcmsg);
		outmsg.msg_flags = 0;

		cmsg = CMSG_FIRSTHDR(&outmsg);
		cmsg->cmsg_level = IPPROTO_SCTP;
		cmsg->cmsg_type = SCTP_SNDRCV;
		cmsg->cmsg_len = CMSG_LEN(sizeof(struct sctp_sndrcvinfo));

		outmsg.msg_controllen = cmsg->cmsg_len;
		memcpy(CMSG_DATA(cmsg), sinfo, sizeof(struct
sctp_sndrcvinfo));
	}

	return sendmsg(s, &outmsg, flags);
}

/*fin funciones*/

int main(int argc, char *argv[1]) {
	int sockfd;
	int len;
	struct sockaddr_in serv_addr;
	struct sockaddr_in *addresses;
	int addr_size = sizeof(struct sockaddr_in);
	int addr_count = argc - 1;
	int port = ECHO_PORT;

	char *message = "Enviando mensaje SCTP al servidor\n";
	struct sctp_initmsg initmsg;
	struct sctp_status status;
	struct sctp_sndrcvinfo sinfo;

	int ochannel;

	if (argc != 5) usage();

	/* create endpoint */

	sockfd = socket(AF_INET, SOCK_STREAM,IPPROTO_SCTP);

	if (sockfd < 0) {
		perror (NULL);
		exit(2);
	}

	/* connect to server */
	addresses = malloc(addr_size);

	if (addresses == NULL) {
		exit(1);
	}
```

```
      serv_addr.sin_family = AF_INET;
      serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
      serv_addr.sin_port = htons(port);

      memcpy(addresses, &serv_addr, addr_size);

      memset(&initmsg, 0, sizeof(initmsg));
      initmsg.sinit_max_instreams = atoi(argv[2]);
      initmsg.sinit_num_ostreams = atoi(argv[3]);
      printf("Negociando numero de streams...\n");
      printf("Cliente tiene disponible: input streams: %d, output
streams: %d\n",
            initmsg.sinit_max_instreams, initmsg.sinit_num_ostreams);

      if (setsockopt(sockfd, IPPROTO_SCTP, SCTP_INITMSG, &initmsg,
sizeof(initmsg))) {
            perror("set sock opt\n");
      }
      if (sctp_connectx(sockfd, (struct sockaddr *) addresses, 1) <
0){
            perror("connectx\n");
            exit(3);
      }

      memset(&status,0,sizeof(status));
      len=sizeof(status);
      status.sstat_assoc_id = 1;

      if (getsockopt(sockfd, IPPROTO_SCTP, SCTP_STATUS, &status, &len)
== -1) {
            perror("get sock opt/n");
      }

      printf("Se llega al acuerdo:\n input  streams: %d, output
streams: %d\n", status.sstat_instrms, status.sstat_outstrms);

      /* sanity check channel */

      ochannel = atoi(argv[4]);

      if (ochannel >= status.sstat_outstrms)
            printf("writing on illegal channel %d\n", ochannel);

      /* transfer data */

      bzero(&sinfo, sizeof(sinfo));
      sinfo.sinfo_stream = ochannel;
      sctp_send(sockfd, message, strlen(message), &sinfo, 0);
      sinfo.sinfo_flags = SCTP_EOF;
      sctp_send(sockfd, NULL, 0, &sinfo, 0);

      close(sockfd);
      exit(0);
}
```

streamsend_echoserver.c

tomado de Newmarch Jan, Linux Journal October 2007

```
 #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <netinet/sctp.h>

#define SIZE 1024
char buf [SIZE];
#define TIME_PORT 2013


/*include extras*/
#include <string.h>
#include <errno.h>
/*fin extra include*/
/*Funciones*/
int sctp_recvmsg(int s, void *msg, size_t len, struct sockaddr *from,
             socklen_t *fromlen, struct sctp_sndrcvinfo *sinfo,
             int *msg_flags)
{
      int error;
      struct iovec iov;
      struct msghdr inmsg;
      char incmsg[CMSG_SPACE(sizeof(struct sctp_sndrcvinfo))];
      struct cmsghdr *cmsg = NULL;

      memset(&inmsg, 0, sizeof (inmsg));

      iov.iov_base = msg;
      iov.iov_len = len;

      inmsg.msg_name = from;
      inmsg.msg_namelen = fromlen ? *fromlen : 0;
      inmsg.msg_iov = &iov;
      inmsg.msg_iovlen = 1;
      inmsg.msg_control = incmsg;
      inmsg.msg_controllen = sizeof(incmsg);

      error = recvmsg(s, &inmsg, 0);
      if (error < 0)
            return error;

      if (fromlen)
            *fromlen = inmsg.msg_namelen;
      if (msg_flags)
            *msg_flags = inmsg.msg_flags;

      if (!sinfo)
            return error;

      for (cmsg = CMSG_FIRSTHDR(&inmsg); cmsg != NULL;
                        cmsg = CMSG_NXTHDR(&inmsg, cmsg)){
            if ((IPPROTO_SCTP == cmsg->cmsg_level) &&
                (SCTP_SNDRCV == cmsg->cmsg_type))
                  break;
```

```
	}

	  /* Copy sinfo. */
	if (cmsg)
		memcpy(sinfo, CMSG_DATA(cmsg), sizeof(struct
sctp_sndrcvinfo));

	return (error);
}
int
sctp_bindx(int fd, struct sockaddr *addrs, int addrcnt, int flags)
{
	int setsock_option = 0;
	void *addrbuf;
	struct sockaddr *sa_addr;
	socklen_t addrs_size = 0;
	int i;

	switch (flags) {
	case SCTP_BINDX_ADD_ADDR:
		setsock_option = SCTP_SOCKOPT_BINDX_ADD;
		break;
	case SCTP_BINDX_REM_ADDR:
		setsock_option = SCTP_SOCKOPT_BINDX_REM;
		break;
	default:
		errno = EINVAL;
		return -1;
	}

	addrbuf = addrs;
	for (i = 0; i < addrcnt; i++) {
		sa_addr = (struct sockaddr *)addrbuf;
		switch (sa_addr->sa_family) {
		case AF_INET:
			addrs_size += sizeof(struct sockaddr_in);
			addrbuf += sizeof(struct sockaddr_in);
			break;
		case AF_INET6:
			addrs_size += sizeof(struct sockaddr_in6);
			addrbuf += sizeof(struct sockaddr_in6);
			break;
		default:
			errno = EINVAL;
			return -1;
		}
	}

	return setsockopt(fd, SOL_SCTP, setsock_option, addrs,
addrs_size);
}

/*fin funciones*/

char *usage_msg = "Usar: ./app direccionIP numero_streams_input
numero_streams_output\n";
void usage() {
	fprintf(stderr, "%s\n", usage_msg);
}
int main(int argc, char *argv[]) {
	int sockfd, client_sockfd;
```

```
      int nread, len;
      struct sockaddr_in serv_addr, client_addr;
      int addr_count=1;
      int port = TIME_PORT;
      struct sctp_initmsg initmsg;
      struct sctp_status status;
      struct sctp_sndrcvinfo sinfo;
      int flags;
      if (argc != 4) usage();

      /* create endpoint */
      sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_SCTP);
      if (sockfd < 0) {
            perror(NULL);
            exit(2);
      }
      serv_addr.sin_family = AF_INET;
      serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
      serv_addr.sin_port = htons(port);

      if (sctp_bindx(sockfd, (struct sockaddr *) &serv_addr,
addr_count,
      SCTP_BINDX_ADD_ADDR)== -1) {
            perror("sctp bindx");
            exit(2);
      }
      memset(&initmsg,0, sizeof(initmsg));
      initmsg.sinit_max_instreams = atoi(argv[2]);
      initmsg.sinit_num_ostreams = atoi(argv[3]);
      printf("Negociando numero de streams...\n");
      printf("Servidor tiene disponible: %d, output atreams; %d\n",
      initmsg.sinit_max_instreams,
      initmsg.sinit_num_ostreams);
      if (setsockopt(sockfd, IPPROTO_SCTP,
      SCTP_INITMSG, &initmsg, sizeof(initmsg))) {
            perror("set sock opt\n");
      }
      /* specity queue,*/
      listen(sockfd, 5);
      for (;;) {
            len = sizeof(client_addr);
            client_sockfd = accept (sockfd, (struct sockaddr *)
&client_addr,
            &len);

            if (client_sockfd == -1) {
                  perror(NULL); continue;
            }

            memset(&status, 0, sizeof(status));
            len = sizeof(status);
            status.sstat_assoc_id= 0;

            if (getsockopt(client_sockfd, IPPROTO_SCTP, SCTP_STATUS,
&status, &len) == -1){
                  perror("get sock opt/n");
            }
            printf("Se llega al acuerdo:\n input streams:   %d, output
streams: %d\n",
            status.sstat_instrms,
            status.sstat_outstrms);
```

```
            for(;;) {
                  /* tranfer data */
                  len = sizeof(client_addr);
                  bzero(&sinfo,sizeof(sinfo));
                  nread = sctp_recvmsg(client_sockfd, buf, SIZE,
                  (struct sockaddr *) &client_addr, &len,   &sinfo,
&flags);

                  if (nread == 0) {
                        break;
                  }
                  printf("read %d bytes on channel %hd\n", nread,
sinfo.sinfo_stream);
                  printf("sinfo flags: %d\n", sinfo.sinfo_flags);
                  write(1, buf, nread);
            }
            close(client_sockfd);
      }
}
```

GeneraPaquetes.c

```
/*************************************************************
   GeneraPaquetes.c - Aplicacion que genera y recibe los paquetes
Traduccion del codigo de Gustavo Do Carmo
*************************************************************/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netinet/sctp.h>
#include <arpa/inet.h>
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include <pthread.h>
#include "./Misc.c"

//-------------------------------------------------------------
//Estruturas de parametros para funciones - threads
//-------------------------------------------------------------
struct parametroT1
{
  int sock_fd;
  struct sockaddr_in direccionIP;
};

//-------------------------------------------------------------
//Protótipos
//-------------------------------------------------------------
//Funciones para control general para mostrar los resultados
void uso (void);
void verificaEntrada (void);
void parser (int argc, char **argv);
void calculaTiempo (void);
void geraScriptGnuPlot (void);

//Funciones para la creacion de la comunicaion
void creaConexionTCP (void);
void creaAsociacionSCTP (void);

//Funcion para envio y recepcion de mensajes
void *enviaMensajesTCP (struct parametroT1 *);
void *enviaMensajesSCTP (struct parametroT1 *);
void *recibeMensajesTCP (struct parametroT1 *);
void *recibeMensajesSCTP (struct parametroT1 *);

//Funciones especificas para TCP
int writeN (int fd, const void *buffer, int n);
int readN (int fd, void *buffer, int n);

//-------------------------------------------------------------
//Variables globales
```

```
//-------------------------------------------------------------
int flujos = 0;
int fila = 0;
char *direccionIPLocal = NULL;
char *direccionIPRemoto = NULL;
int puertoLocal = 0;
int puertoRemoto = 0;
int tamanoBuffer = 0;
int numeroMensajesEnviar = 30;
int tipoDeProtocolo = 0;
int tamanoMensaje = 20;
int mensajeDesordenado = 0;
int intervaloEntreMensajes = 0;
int verbose = 1;
FILE *archivo;
FILE *archivo2;

//Representacion de mensajes Gnutella
extern char *gnutella[5];

//estrutura enviada via socket
struct tipao
{
  int id;
  char sendline[100];
};

//Estruturas y variables para el calculo de tiempo
struct paraContar
{
  int id;
  int tamano;
  struct timeval cronometroIda;
  struct timeval cronometroVuelta;
};

struct timeval tiempoGlobalInicial;
struct timeval tiempoGlobalFinal;
int cantidadEnviadaGlobal = 0;
struct paraContar matriz[NUMERO_MAX_MENSAJES];

//Miscelania
double latenciaMedia;
double velocidadMedia;
double velocidadGeneral = 0;
double latenciaGeneral = 0;
double latenciaTotal;
double velocidadTotal;
double latenciaIndividual[NUMERO_MAX_MENSAJES];
double velocidadIndividual[NUMERO_MAX_MENSAJES];

//-------------------------------------------------------------
//Programa principal
//-------------------------------------------------------------
int
main (int argc, char **argv)
{
  parser (argc, argv);
  verificaEntrada ();

  //Archivos para almacenar los dados de desempeno
```

```
  char nombrearchivo1[50];
  char nombrearchivo2[50];

  sprintf (nombrearchivo1,
         "./ResultadosLatencias_T%d_M%d_S%d_F%d_D%d_U%d.txt",
         tipoDeProtocolo, numeroMensajesEnviar, tamanoMensaje,
         flujos, mensajeDesordenado, intervaloEntreMensajes);

  sprintf (nombrearchivo2,
         "./ResultadosVelocidad_T%d_M%d_S%d_F%d_D%d_U%d.txt",
         tipoDeProtocolo, numeroMensajesEnviar, tamanoMensaje,
         flujos, mensajeDesordenado, intervaloEntreMensajes);

  archivo = fopen (nombrearchivo1, "w");
  archivo2 = fopen (nombrearchivo2, "w");

  int i;

  //Ejecuta el programa por un número definido de veces
  if (tipoDeProtocolo == 1)
    for (i = 1; i <= verbose; i++)
      {                        //Protocolo escojido fue TCP
      creaConexionTCP ();
      }
  else
    for (i = 1; i <= verbose; i++)
      {                        //Protocolo escojido fue SCTP
      creaAsociacionSCTP ();
      }

  fclose (archivo);
  fclose (archivo2);

  return 0;
}                       //fin de main


//------------------------------------------------------------------
//Adquiriendo los datos de la linea de comandos
//------------------------------------------------------------------
void
parser (int argc, char **argv)
{
  int getoptC;
  int bufferAux;
  int numAux;
  int tamanoAux;
  int desorden;
  int intervalo;
  int verb;

  while (argc)
    {
      getoptC = getopt (argc, argv, "H:P:h:p:t:x:f:b:m:s:d:u:v:");

      if (getoptC < 0)
      break;

      switch (getoptC)
      {
```

```
case 'H':
  direccionIPLocal = optarg;
  break;

case 'P':
  puertoLocal = atoi (optarg);
  break;

case 'h':
  direccionIPRemoto = optarg;
  break;

case 'p':
  puertoRemoto = atoi (optarg);
  break;

case 't':
  tipoDeProtocolo = atoi (optarg);
  break;

case 'x':
  flujos = atoi (optarg);
  break;

case 'f':
  fila = atoi (optarg);
  break;

case 'b':
  bufferAux = atoi (optarg);
  if ((bufferAux > 0) && (bufferAux < 1001))
    tamanoBuffer = bufferAux;
  break;

case 'm':
  numAux = atoi (optarg);
  if ((numAux > 0) && (numAux < 101))
    numeroMensajesEnviar = numAux;
  break;

case 's':
  tamanoAux = atoi (optarg);
  if ((tamanoAux > 1) && (tamanoAux < 101))
    tamanoMensaje = tamanoAux;
  break;

case 'd':
  desorden = atoi (optarg);
  if (desorden == 1)
    mensajeDesordenado = MSG_UNORDERED;
  break;

case 'u':
  intervalo = atoi (optarg);
  if ((intervalo > 0) && (intervalo < 1001))
    intervaloEntreMensajes = intervalo;
  break;

case 'v':
  verb = atoi (optarg);
  if ((verb > 1) && (verb < 101))
```

```
          verbose = verb;
        break;

      default:
        break;
      }                 //fin switch
    }                   //fin while argc
}                       //fin de parser


//---------------------------------------------------------------
//Revisando los datos de la linea de comandos
//---------------------------------------------------------------
void
verificaEntrada (void)
{
  if ((direccionIPLocal == NULL) ||
      (inet_aton (direccionIPLocal, NULL) == 0))
    {
      printf ("\nFalto colocar una direccion IP local\n");
      uso ();
      exit (0);
    }

  if (puertoLocal == 0)
    {
      printf ("\nFalto colocar un puerto local\n");
      uso ();
      exit (0);
    }

  if ((direccionIPRemoto == NULL) ||
      (inet_aton (direccionIPRemoto, NULL) == 0))
    {
      printf ("\nFalto colocar una direccion IP Remoto\n");
      uso ();
      exit (0);
    }

  if (puertoRemoto == 0)
    {
      printf ("\nFalto colocar un puerto Remoto\n");
      uso ();
      exit (0);
    }

  if ((tipoDeProtocolo != 1) && (tipoDeProtocolo != 2))
    {
      printf ("\nEscoja el protocolo!\n");
      printf ("\nTCP=1 o SCTP=2\n");
      uso ();
      exit (0);
    }

  if ((flujos < 1) && (tipoDeProtocolo == 2))
    {
      printf ("\nFluJo configurado para 1\n");
      flujos = 1;
    }

  if ((fila < 1) && (tipoDeProtocolo == 2))
```

```
    {
      printf ("\nFila seteada para 1\n");
      fila = 1;
    }

  printf ("\n");
  printf ("DireccionIP del host local:%s\n", direccionIPLocal);
  printf ("Puerto del host local:%d\n", puertoLocal);
  printf ("DireccionIP del host Remoto:%s\n", direccionIPRemoto);
  printf ("Puerto del host Remoto:%d\n", puertoRemoto);

  if ((tipoDeProtocolo == 1))       //Caso TCP
    printf ("Protocolo de transporte: TCP\n");
  else
    {                   //Caso SCTP
      printf ("Protocolo de transporte: STCP\n");
      printf ("Numero de flujos:%d\n", flujos);
      printf ("Fila de Mensajes en cada flujo:%d\n", fila);

      if (tamanoBuffer == 0)
      printf ("Buffer de envio y recepcion configurado como
patron\n");
      else
      printf ("Buffer de envio y recepcion (kBytes):%d\n",
            tamanoBuffer);

      if (mensajeDesordenado == 0)
      printf ("Entrega de Mensajes: ordenada\n");
      else
      printf ("Entrega de Mensajes: desordenada\n");
    }                   //Fin caso SCTP

  printf ("Numero de Mensajes a enviar:%d\n",
        numeroMensajesEnviar);
  printf ("tamano de Mensajes (bytes):%d\n", tamanoMensaje);
  printf ("Intervalo entre Mensajes (milisegundos):%d\n",
        intervaloEntreMensajes);
  printf ("\n");
}                       //fin verificaEntrada


//---------------------------------------------------------------
//Imprime la forma de uso de la aplicacion o help
//---------------------------------------------------------------
void
uso (void)
{
  printf ("\n");
  printf ("Uso: ./GeneraPaquetes -H dirIPLoc -P ptLoc -h dirIPRem -p
ptRem -t tipProt \n");
  printf ("                  -x numFlujos -f fila -b buffer -m Mensajes
-s size\n");
  printf ("                  -d orden -u tiempo -v verbose\n\n");
  printf ("    -H dirIPLoc        - direccion IP del host local\n");
  printf ("    -P ptLoc          - puerto del host local\n");
  printf ("    -h dirIPRem         - direccion IP del host Remoto\n");
  printf ("    -p ptRem          - puerto del host Remoto\n");
  printf ("    -t tipProt        - protocolo de transporte - TCP=1 -
SCTP=2\n");
  printf ("    -x numFlujos        - número de flujos - min 1 - max
300 - SCTP\n");
```

```
  printf ("    -f fila           - fila de Mensajes en cada flujo -
min 1 - max 5 - SCTP\n");
  printf ("    -b buffer         - buffer de envio y recibo (kBytes)
");
  printf ("- min 1 - max 1000 - SCTP\n");
  printf ("    -m Mensajes      - número de Mensajes a enviar - min 1
- max 100\n");
  printf ("    -s size           - tamano da mensaje (bytes) - min 2 -
max 100\n");
  printf ("    -d orden          - 1 - envia los Mensajes de manera
desordenada\n");
  printf ("    -u tiempo          - tiempo entre Mensajes -
milisegundos - patron 50\n");
  printf ("    -v verbose        - verbose 1 o 2\n");
  printf ("\n");
}                       //fin help

//-------------------------------------------------------------------
//-------------------------------------------------------------------
void
creaAsociacionSCTP (void)
{
  int sock_fd;
  int erro;

  struct sockaddr_in servaddr, meuaddr, direccionIPEnviante;
  struct sctp_event_subscribe evnts;

  //Creacion del socket
  sock_fd = socket (AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);

  if (sock_fd < 0)
    {
      printf ("Fallo creacion del socket SCTP\n");
      exit (0);
    }

  //Configurando las direcciones IP
  bzero (&servaddr, sizeof (servaddr));
  servaddr.sin_family = AF_INET;
  inet_aton (direccionIPRemoto, &servaddr.sin_addr);
  servaddr.sin_port = htons (puertoRemoto);

  bzero (&meuaddr, sizeof (meuaddr));
  meuaddr.sin_family = AF_INET;
  inet_aton (direccionIPLocal, &meuaddr.sin_addr);
  meuaddr.sin_port = htons (puertoLocal);

  // notificaciones de eventos(Solo en SCTP)
  bzero (&evnts, sizeof (evnts));
  evnts.sctp_data_io_event = 1;

  erro = setsockopt (sock_fd, IPPROTO_SCTP, SCTP_EVENTS,
                &evnts, sizeof (evnts));

  if (erro != 0)
    {
      printf ("Error al configurar la opcion de initmessage en el
socket\n");
      exit (0);
    }
```

```
  int desliga = 1;

  //Desactivando el algoritmo Nagle
  erro = setsockopt (sock_fd, IPPROTO_SCTP, SCTP_NODELAY,
                &desliga, sizeof (desliga));

  if (erro == (-1))
    {
      printf ("Error al desconectar el algoritmo Nagle en SCTP\n");
      exit (0);
    }

  struct sctp_initmsg initmsg;

  //Configurando el número de flujos
  initmsg.sinit_num_ostreams = flujos;
  initmsg.sinit_max_instreams = flujos;
  initmsg.sinit_max_attempts = 3;
  initmsg.sinit_max_init_timeo = 30;

  erro = setsockopt (sock_fd, IPPROTO_SCTP, SCTP_INITMSG,
                (char *) &initmsg,
                sizeof (struct sctp_initmsg));

  if (erro != 0)
    {
      printf ("Error al configurar la opcion de initmessage en el
socket\n");
      exit (0);
    }

  //Configurando el tamaño del buffer
  tamanoBuffer = tamanoBuffer * 1000;

  if (tamanoBuffer != 0)
    {
      erro = setsockopt (sock_fd, IPPROTO_SCTP, SO_RCVBUF,
                   (char *) &tamanoBuffer,
                   sizeof (tamanoBuffer));

      if (erro != 0)
      {
        printf ("Error al configurar la opcion de recepcion de buffer
en el socket\n");
        exit (0);
      }

      erro = setsockopt (sock_fd, IPPROTO_SCTP,
                   SO_SNDBUF, (char *) &tamanoBuffer,
                   sizeof (tamanoBuffer));

      if (erro != 0)
      {
        printf ("Error al configurar la opcion de envio de buffer en
el socket\n");
        exit (0);
      }
    }                   //fin tamanobuffer!=0

  //Creacion del bind
```

```
  erro = bind (sock_fd, (struct sockaddr *) &meuaddr,
               sizeof (meuaddr));

  if (erro != 0)
    {
      printf ("Error en la creacion de la funcion bind\n");
      exit (0);
    }

  //Esta parte es para contar el tiempo del establecimiento de la
asociacion SCTP
  //es decir, la latencia.
  struct sctp_sndrcvinfo sri;
  struct tipao mensaje;
  int cantidadEnviada;
  int cantidadRecibida;
  int msg_flags;

  bzero (&sri, sizeof (sri));
  sri.sinfo_stream = 0;

  int out_sz;

  socklen_t len;
  len = sizeof (struct sockaddr_in);

  char *sendline;
  sendline = malloc (25);

  bzero (mensaje.sendline, sizeof (100));
  bzero (sendline, sizeof (25));
  sprintf (mensaje.sendline, "Conectando");
  out_sz = strlen (mensaje.sendline) + sizeof (int);
  mensaje.id = 1000;

  cantidadEnviada =
        sctp_sendmsg (sock_fd, "Conectando", out_sz,
                  (struct sockaddr *) &servaddr,
                  sizeof (struct sockaddr), 0,
                  mensajeDesordenado, sri.sinfo_stream, 0,
                  0);

  if (cantidadEnviada == (-1))
    {
      printf ("Falla al enviar mensaje\n");
    }

  cantidadRecibida =
        sctp_recvmsg (sock_fd, sendline, 25,
                  (struct sockaddr *) &direccionIPEnviante,
                  &len, &sri, &msg_flags);

  if (cantidadRecibida == (-1))
    {
      printf ("Falla al recibir mensaje\n");
    }

  //Fin     del establecimiento de la asociacion SCTP

  struct parametroT1 valores1;
  struct parametroT1 valores2;
```

```
  //Llenando las estruturas de los parametros de los threads
  valores1.sock_fd = sock_fd;
  valores1.direccionIP = direccionIPEnviante;
  valores2.sock_fd = sock_fd;
  valores2.direccionIP = servaddr;

  pthread_t p_thread_envia;
  pthread_t p_thread_recibe;

  //Abre un thread que genera y envia mensajes
  pthread_create (&p_thread_envia, NULL,
              (void *) enviaMensajesSCTP, &valores2);

  //Abre un thread que recibe mensajes
  pthread_create (&p_thread_recibe, NULL,
              (void *) recibeMensajesSCTP, &valores1);

  //Guarda los términos de ambos threads
  pthread_join (p_thread_envia, NULL);
  pthread_join (p_thread_recibe, NULL);

  //Calcula y muestra resultados
  calculaTiempo ();
  geraScriptGnuPlot ();

  return;
}                       //fin de creacion de la asociacion SCTP

//------------------------------------------------------------------
//------------------------------------------------------------------
void
geraScriptGnuPlot (void)
{
  printf ("\n");
  printf ("#-----------------------------------------------------------
--------\n");
  printf ("#SCRIPT PARA GNUPLOT\n");
  printf ("#-----------------------------------------------------------
--------\n");

  printf ("\n");
  printf ("#-----------------------------------------------------------
--------\n");
  printf ("#MANUAL DE LECTURA\n");
  printf ("#-----------------------------------------------------------
--------\n");

  printf ("\n");
  printf ("#-latencia media: la media aritmética de todas las
latencias individuales\n");
  printf ("#-velocidad media:    la media aritmética de todas las
velocidades individuales\n");
  printf ("\n");
  printf ("#-latencia General: la suma de todas las latencias, sin
considerar los intervalos\n");
  printf ("#-velocidad General:    la suma de todas las velocidades,
sin considerar los intervalos\n");
  printf ("\n");
  printf ("#-latencia total: desde el primer mensaje enviado hasta el
ultimo recibido\n");
```

```
  printf ("#-velocidad total:    desde el primer mensaje enviado hasta
el ultimo recibido\n");

  printf ("\n");
  printf ("#-------------------------------------------------------------
--------\n");
  printf ("#INFORMACIOPN GENERAL\n");
  printf ("#-------------------------------------------------------------
--------\n");

  printf ("\n");
  printf ("#Direccion IP del host local:%s\n", direccionIPLocal);
  printf ("#Puerto del host local:%d\n", puertoLocal);
  printf ("#Direccion IP del host Remoto:%s\n", direccionIPRemoto);
  printf ("#Puerto del host Remoto:%d\n", puertoRemoto);

  if ((tipoDeProtocolo == 1)) //Caso TCP
    printf ("#Protocolo de transporte: TCP\n");
  else
    {                   //Caso SCTP
      printf ("#Protocolo de transporte: STCP\n");
      printf ("#Número de flujos:%d\n", flujos);
      printf ("#Fila de Mensajes en cada flujo:%d\n", fila);

      if (tamanoBuffer == 0)
      printf ("#Buffer de envio y recepcion configurado como
patron\n");
      else
      printf ("#Buffer de envio y recepcion (kBytes):%d\n",
            tamanoBuffer);
    }                   //Fin caso SCTP

  printf ("#Número de Mensajes a enviar:%d\n",
        numeroMensajesEnviar);
  printf ("#tamano de Mensajes (bytes):%d\n", tamanoMensaje);
  printf ("\n");
  printf ("#Round Trip Time:\n");
  printf ("#Tasa de perdida de paquetes:\n");
      printf ("\n");

  printf ("Latencia Media      %0.6f \n", latenciaMedia);
  printf ("Velocidad Media     %0.6f \n", velocidadMedia);
  printf ("Latencia General    %0.6f \n", latenciaGeneral);
  printf ("Velocidad General   %0.6f \n", velocidadGeneral);
  printf ("Latencia Total      %0.6f \n", latenciaTotal);
  printf ("Velocidad Total     %0.6f \n", velocidadTotal);
      printf ("\n");
  printf ("#--------------------------------------\n");
  printf ("#IdMsg:       Lat(s):     Vel(b/s):    \n");
  printf ("#--------------------------------------\n");

  printf ("\n");

  //printf ("#-------------------------------------------------------------
----------\n");
  //printf ("#Lat General(s) : VelGeneral(b/s) : Lat Total(s) :
VelTotal(b/s)\n");
  //printf ("#-------------------------------------------------------------
----------\n");
  //printf ("\n");
```

```
  int i;

  for (i = 0; i < numeroMensajesEnviar; i++)
    {
      printf ("%d         %0.6f    %0.6f\n", i, latenciaIndividual[i],
velocidadIndividual[i], latenciaMedia, velocidadMedia, latenciaGeneral,
velocidadGeneral, latenciaTotal, velocidadTotal);
    }



  fprintf (archivo, "%0.6f ", latenciaMedia);
  fprintf (archivo2, "%0.6f ", velocidadMedia);
}                        //fin de generaScript


//---------------------------------------------------------------------
//---------------------------------------------------------------------
void
calculaTiempo (void)
{
  int i, j;
  double resultadotiempoSec;
  double resultadotiempoMicro;
  double resultadoFinal;
  double velocidad;


  printf ("\n");
  printf ("-----------------------------------------------------\n");
  printf ("CÁLCULO DE LATENCIA Y VELOCIDAD POR MENSAJE ENVIADO\n");
  printf ("-----------------------------------------------------\n");

  for (i = 0; i < numeroMensajesEnviar; i++)
    {
      for (j = 0; j < numeroMensajesEnviar; j++)
      {
        if (i == matriz[j].id)
          {
            resultadotiempoSec =
                  (matriz[j].cronometroVuelta.tv_sec) -
                  (matriz[j].cronometroIda.tv_sec);

            resultadotiempoMicro =
                  (double) (matriz[j].cronometroVuelta.
                        tv_usec) -
                  (double) (matriz[j].cronometroIda.tv_usec);

            resultadoFinal =
                  resultadotiempoSec +
                  ((double) resultadotiempoMicro /
                   (double) 1000000);

            printf ("\n");
            printf ("Latencia por mensaje %d = %0.6f segundos\n", i,
resultadoFinal);

            latenciaIndividual[i] = resultadoFinal;

            velocidad = (double) (matriz[j].tamano) /
                  (double) resultadoFinal;
```

```
            printf ("Velocidad por mensaje %d = %0.6f bytes/seg\n",
                  i, velocidad);

            printf ("Cantidad de datos enviado: %d\n",
                  matriz[j].tamano);

            velocidadIndividual[i] = velocidad;

            j = numeroMensajesEnviar;
          }             //fin if
      }                 //fin for 2
    }                   //fin for 1

  //Cálculo da latencia y taza de transferencia globales
  resultadotiempoSec =
        (tiempoGlobalFinal.tv_sec) - (tiempoGlobalInicial.tv_sec);

  resultadotiempoMicro =
        (double) (tiempoGlobalFinal.tv_usec) -
        (double) (tiempoGlobalInicial.tv_usec);

  resultadoFinal =
        resultadotiempoSec +
        ((double) resultadotiempoMicro / (double) 1000000);

  printf ("\n");
  printf ("-----------------------------------------------------\n");
  printf ("\n");
  printf ("Latencia Total Final = %0.6f segundos\n",
        resultadoFinal);

  latenciaTotal = resultadoFinal;

  velocidad = (double) (cantidadEnviadaGlobal) /
        (double) resultadoFinal;

  printf ("Cantidad de datos enviado: %d bytes\n",
        cantidadEnviadaGlobal);

  printf ("Velocidad Total Final = %0.6f bytes/seg\n\n", velocidad);

  velocidadTotal = velocidad;

  for (i = 0; i < numeroMensajesEnviar; i++)
    {
      latenciaGeneral = latenciaIndividual[i] + latenciaGeneral;

      velocidadGeneral = velocidadIndividual[i] + velocidadGeneral;
    }

  printf ("Latencia General = %0.6f seg\n\n", latenciaGeneral);

  latenciaMedia =
        (double) latenciaGeneral / (double) numeroMensajesEnviar;

  printf ("Latencia Media = %0.6f seg\n\n", latenciaMedia);

  printf ("Velocidad General = %0.6f bytes/seg\n\n",
velocidadGeneral);
```

```
  velocidadMedia =
        (double) velocidadGeneral / (double) numeroMensajesEnviar;

  printf ("Velocidad Media = %0.6f bytes/seg\n\n", velocidadMedia);

}                       //fin de calculaTiempo


//-------------------------------------------------------------
//Funcion que genera y envia los mensajes a usando SCTP
//-------------------------------------------------------------
void *
enviaMensajesSCTP (struct parametroT1 *variables)
{
  printf ("Se ejecuta la funcion de enviar mensajes\n");

  struct tipao mensaje;

  int sock_fd = variables->sock_fd;
  struct sockaddr_in servaddr = variables->direccionIP;

  int loop;
  int indiceFlujos = 0;
  int out_sz;
  int indiceFila = fila;
  int indiceMensaje = 0;
  int cantidadEnviada;
  struct sctp_sndrcvinfo sri;

  //Esta estructura es usada para ver en que flujo vino el mensaje

  bzero (&sri, sizeof (sri));
  sri.sinfo_stream = 0;

  printf ("\n");
  printf ("-----------------------------------------------------\n");
  printf ("DADOS SOBRE EL ENVIO DE MENSAJES\n");
  printf ("-----------------------------------------------------\n");

  //loop de envio de Mensajes
  for (loop = 0; loop < numeroMensajesEnviar; loop++)
    {
      sri.sinfo_stream = indiceFlujos;
      bzero (mensaje.sendline, sizeof (100));
      snprintf (mensaje.sendline, tamanoMensaje,
            gnutella[loop % 5]);
      out_sz = strlen (mensaje.sendline) + 1;
      mensaje.id = loop;

      printf ("\n");
      printf ("ENVIO\n");
      printf ("ID Mensaje:%d\n", mensaje.id);
      printf ("Mensaje:%s\n", mensaje.sendline);
      printf ("Numero Flujo:%d\n", indiceFlujos);
      printf ("Indice Fila:%d\n", indiceFila);
      printf ("Numero Loop:%d\n", loop);

      //Tiempo entre los mensajes en milisegundos
      usleep (1000 * intervaloEntreMensajes);

      //Início del calculo del tiempo de ida  del mensaje
```

```
      matriz[loop].id = mensaje.id;

      gettimeofday (&matriz[loop].cronometroIda, NULL);

      //Enviando un mensaje
      cantidadEnviada =
            sctp_sendmsg (sock_fd, &mensaje, out_sz,
                      (struct sockaddr *) &servaddr,
                      sizeof (struct sockaddr), 0,
                      mensajeDesordenado,
                      sri.sinfo_stream, 0, 0);

      if (cantidadEnviada == (-1))
      {
        printf ("Falla al enviar mensaje\n");
      }

      //Para cálculo del total de bytes enviados
      cantidadEnviadaGlobal =
            cantidadEnviadaGlobal + cantidadEnviada;

      printf ("Bytes enviados:%d\n", cantidadEnviada);
      matriz[loop].tamano = cantidadEnviada;

      //Control de la fila en los flujos
      if (indiceFila == 1)
      {
        if (indiceFlujos == (flujos - 1))
          {
            indiceFlujos = 0;
          }
        else
          {
            indiceFlujos++;
          }
        indiceFila = fila;
      }
      else
      indiceFila--;

      if (indiceMensaje == 4)
      indiceMensaje = 0;
      else
      indiceMensaje++;
    }                   //fin del loop for de envio de mensajes

  return 0;
}                       //fin de envioMensajes


//------------------------------------------------------------------
//Funcion que recibe los mensajes usando SCTP
//------------------------------------------------------------------
void *
recibeMensajesSCTP (struct parametroT1 *variables)
{
  printf ("Se ejecuta la funcion de recibir mensajes\n");

  //Utilizado para recibir los mensajes
  struct tipao mensaje;
```

```
//Llenando las variables locales con los datos del parametro
int sock_fd = variables->sock_fd;
struct sockaddr_in direccionIPEnviante = variables->direccionIP;

int loop;
int cantidadRecibida;
struct sctp_sndrcvinfo sri;

char *sendline;
sendline = malloc (105);

socklen_t len;
len = sizeof (struct sockaddr_in);
int msg_flags;

//Estrutura utilizada para saber en flujo vino el mensaje
bzero (&sri, sizeof (sri));
sri.sinfo_stream = 0;

printf ("\n");
printf ("-----------------------------------------------------\n");
printf ("DADOS SOBRE A RECEPCION DE MENSAJES\n");
printf ("-----------------------------------------------------\n");

//loop de recepcion de mensajes
for (loop = 0; loop < numeroMensajesEnviar; loop++)
  {
    bzero (sendline, 105);

    //Recibiendo un mensaje
    cantidadRecibida =
          sctp_recvmsg (sock_fd, sendline, 105,
                    (struct sockaddr *) &direccionIPEnviante,
                    &len, &sri, &msg_flags);

    //Armacenando el momento de llegada del mensaje
    gettimeofday (&matriz[loop].cronometroVuelta, NULL);

    usleep (1);

    bzero (mensaje.sendline, 100);
    memcpy (&mensaje, sendline, cantidadRecibida);

    matriz[loop].id = mensaje.id;

    if (cantidadRecibida == (-1))
    {
      printf ("Falla al recibir mensaje\n");
      loop = loop - 1;
      continue;
    }

    if (loop == numeroMensajesEnviar - 1)
    {
      tiempoGlobalInicial = matriz[0].cronometroIda;
      tiempoGlobalFinal = matriz[loop].cronometroVuelta;
    }

    //Impresion de datos de la recepcion
    printf ("\n");
    printf ("RECIBI\n");
```

```
      printf ("ID=%d - Mensaje:%s\n",
            mensaje.id, mensaje.sendline);
      printf ("Numero Flujo:%d\n", sri.sinfo_stream);
      printf ("Cantidad Recibida:%d\n", cantidadRecibida);

    }                   //fin loop de recepcion de mensajes

  free (sendline);
  close (sock_fd);

  return 0;
}                       //fin recibe mensajes


//---------------------------------------------------------------
//Funcion para enviar N bytes usando TCP
//---------------------------------------------------------------
int
writeN (int fd, const void *buffer, int n)
{
  int nRestante;
  int nEscritos;
  const char *ptr;

  ptr = buffer;
  nRestante = n;

  while (nRestante > 0)
    {
      if ((nEscritos = write (fd, ptr, nRestante)) <= 0)
      {
        if (nEscritos < 0 && errno == EINTR)
          nEscritos = 0;
        else
          return (-1);
      }

      nRestante -= nEscritos;
      ptr += nEscritos;
    }

  return n;

}                       //fin writeN


//---------------------------------------------------------------
//Funcion para recibir N bytes usando TCP
//---------------------------------------------------------------
int
readN (int fd, void *buffer, int n)
{
  int nRestante;
  int nLeidos;
  char *ptr;

  ptr = buffer;
  nRestante = n;

  while (nRestante > 0)
    {
```

```
      if ((nLeidos = read (fd, ptr, nRestante)) < 0)
      {
        if (errno == EINTR)
          nLeidos = 0;
        else
          return (-1);
      }
      else if (nLeidos == 0)
      break;

      nRestante -= nLeidos;
      ptr += nLeidos;
    }

  return (n - nRestante);

}                       //fin readN

//-------------------------------------------------------------------
//Funcion para crear una conexion TCP con EchoServer
//-------------------------------------------------------------------
void
creaConexionTCP (void)
{
  int sock_fd;
  int erro;

  struct sockaddr_in servaddr, meuaddr;

  //Abre un socket
  sock_fd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);

  if (sock_fd < 0)
    {
      printf ("Creacion del socket fallo\n");
      exit (0);
    }

  int desliga = 1;

  //Desliga o Nagle
  erro = setsockopt (sock_fd, IPPROTO_TCP, TCP_NODELAY,
                &desliga, sizeof (desliga));

  if (erro == (-1))
    {
      printf ("Error al desconectar el algoritmo Nagle en TCP\n");

    }

  //Definiendo las Direcciones IP
  bzero (&servaddr, sizeof (servaddr));
  servaddr.sin_family = AF_INET;
  inet_aton (direccionIPRemoto, &servaddr.sin_addr);
  servaddr.sin_port = htons (puertoRemoto);

  bzero (&meuaddr, sizeof (meuaddr));
  meuaddr.sin_family = AF_INET;
  inet_aton (direccionIPLocal, &meuaddr.sin_addr);
  meuaddr.sin_port = htons (puertoLocal);
```

```
  //Intentando conectar al EchoServer
  erro = connect (sock_fd, (struct sockaddr *) &servaddr,
              sizeof (servaddr));

  if (erro < 0)
    {
      printf ("Error al efectuar la conexion\n");
      exit (0);
    }

  struct parametroT1 valores1;
  struct parametroT1 valores2;

  valores1.sock_fd = sock_fd;
  valores2.sock_fd = sock_fd;

  pthread_t p_thread_envia;
  pthread_t p_thread_recibe;

  //Crea un thread que envia mensajes
  pthread_create (&p_thread_envia, NULL,
              (void *) enviaMensajesTCP, &valores2);

  //Crea un thread que recibe mensajes
  pthread_create (&p_thread_recibe, NULL,
              (void *) recibeMensajesTCP, &valores1);

  //Guarda los dos threads terminarem
  pthread_join (p_thread_envia, NULL);
  pthread_join (p_thread_recibe, NULL);

  //Calcula resultados  y los muestra
  calculaTiempo ();
  geraScriptGnuPlot ();

  return;

}                       //fin conexion TCP


//-------------------------------------------------------------
//Funcion para envio de mensajes usando TCP
//-------------------------------------------------------------
void *
enviaMensajesTCP (struct parametroT1 *variables)
{
  struct tipao mensaje;

  int sock_fd = variables->sock_fd;
  int out_sz;
  int loop;
  int cantidadEnviada;

  printf ("\n");
  printf ("-----------------------------------------------------\n");
  printf ("DADOS DE ENVIO DE MENSAJES\n");
  printf ("-----------------------------------------------------\n");

  //loop de envio de mensajes
  for (loop = 0; loop <= numeroMensajesEnviar; loop++)
    {
```

```
      bzero (mensaje.sendline, sizeof (100));
      snprintf (mensaje.sendline, tamanoMensaje,
            gnutella[loop % 5]);
      out_sz = strlen (mensaje.sendline) + 1;
      mensaje.id = loop;

      //Impresion de datos  sobre el envio
      printf ("\n");
      printf ("ENVIO\n");
      printf ("ID Mensaje:%d\n", mensaje.id);
      printf ("Mensaje:%s\n", mensaje.sendline);
      printf ("Número Loop:%d\n", loop);

      //Início do cálculo do tiempo de ida da mensaje
      matriz[loop].id = mensaje.id;
      gettimeofday (&matriz[loop].cronometroIda, NULL);

      usleep (1000 * intervaloEntreMensajes);

      cantidadEnviada = writeN (sock_fd, &mensaje, out_sz);

      if (cantidadEnviada == (-1))
      {
        printf ("Falla al enviar el mensaje\n");
      }

      //Para cálculo del total de bytes enviados
      cantidadEnviadaGlobal =
            cantidadEnviadaGlobal + cantidadEnviada;

      printf ("Bytes enviados:%d\n", cantidadEnviada);
      matriz[loop].tamano = cantidadEnviada;

    }                   //fin del loop for de envio de mensajes

  return 0;

}                       //fin de envia mensajes


//-------------------------------------------------------------
//Funcion que recibe mensajes usando TCP
//-------------------------------------------------------------
void *
recibeMensajesTCP (struct parametroT1 *variables)
{
  struct tipao mensaje;

  int sock_fd = variables->sock_fd;
  int loop;
  socklen_t len;
  int cantidadRecibida;

  len = sizeof (struct sockaddr_in);

  char *sendline;
  sendline = malloc (105);

  printf ("\n");
  printf ("-----------------------------------------------------\n");
  printf ("DADOS DE LA RECEPCION DE MENSAJES\n");
```

```
  printf ("-----------------------------------------------------\n");

  //loop de recepcion de mensajes
  for (loop = 0; loop <= numeroMensajesEnviar; loop++)
    {
      bzero (sendline, 105);
      cantidadRecibida =
            readN (sock_fd, sendline, matriz[loop].tamano);

      gettimeofday (&matriz[loop].cronometroVuelta, NULL);

      usleep (1);

      if (cantidadRecibida == (-1))
      {
        printf ("Falla al recibir mensaje\n");
        loop = loop - 1;
        continue;
      }

      if (cantidadRecibida == (0))
      {
        loop = loop - 1;
        continue;
      }

      bzero (mensaje.sendline, 100);
      memcpy (&mensaje, sendline, cantidadRecibida);

      if (loop == numeroMensajesEnviar - 1)
      {
        tiempoGlobalInicial = matriz[0].cronometroIda;
        tiempoGlobalFinal = matriz[loop].cronometroVuelta;
      }

      //Impresion de datos de la recepcion
      printf ("\n");
      printf ("RECEBI\n");
      printf ("ID=%d - mensaje:%s\n", mensaje.id,
            mensaje.sendline);
      printf ("Cantidad Recibida: %d\n", cantidadRecibida);
      printf ("Cantidad que era para recibir: %d\n",
            matriz[loop].tamano);

    }                   //fin loop de rececion de mensajes

  free (sendline);
  close (sock_fd);

  return 0;
}                       //fin de recibe mensajes

//--------------------------------------------------------------------
//Fin de GeneraPaquetes.c
//--------------------------------------------------------------------
```

EchoServer.c

```
/**************************************************************
         EchoServer.c - Echo Server
Traduccion del codigo de Gustavo Do Carmo

**************************************************************/
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/sctp.h>
#include <netinet/tcp.h>
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

//--------------------------------------------------------------
//Protótipos
//--------------------------------------------------------------
void uso (void);
void parser (int argc, char **argv);
void verificaEntrada (void);
void echoSCTP (void);
void echoTCP (void);
int writeN (int fd, const void *buffer, int n);

//--------------------------------------------------------------
//Variables globales
//--------------------------------------------------------------
int tamanoBuffer = 0;
int puertoLocal = 0;
char *direccionLocal = NULL;
int mensajeDesordenado = 0;
int tipoProtocolo = 1;

//Estrutura usada para recibir mensajes
struct tipao
{
  int id;
  char sendline[100];
} mensagem;

//--------------------------------------------------------------
//Programa principal
//--------------------------------------------------------------
int
main (int argc, char **argv)

  parser (argc, argv);
  verificaEntrada ();

  if (tipoProtocolo == 1)
    echoTCP ();
  else if (tipoProtocolo == 2)
    echoSCTP ();
  else
```

```
    uso ();

  return 0;
}


//------------------------------------------------------------------
//EchoServer usando TCP
//------------------------------------------------------------------
void
echoTCP ()
{
  int sock_fd, sock_nuevo;
  int erro;

  struct sockaddr_in servaddr, cliaddr;

  sock_fd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);

  if (sock_fd < 0)
    {
      printf ("Falla de creacion del socket\n");
      exit (0);
    }

  bzero (&servaddr, sizeof (servaddr));
  servaddr.sin_family = AF_INET;
  inet_aton (direccionLocal, &servaddr.sin_addr);
  servaddr.sin_port = htons (puertoLocal);

  int desconecta = 1;

  erro = setsockopt (sock_fd, IPPROTO_TCP, TCP_NODELAY,
                &desconecta, sizeof (desconecta));

  if (erro == (-1))
    {
      printf ("Error en el socket TCP\n");
    }

  erro = bind (sock_fd, (struct sockaddr *) &servaddr,
            sizeof (servaddr));

  if (erro != 0)
    {
      printf ("Error al crear la funcion bind\n");
      exit (0);
    }

  erro = listen (sock_fd, 1);

  if (erro != 0)
    {
      printf ("Error en la funcion listen\n");
      exit (0);
    }

  socklen_t len;

  len = sizeof (struct sockaddr_in);
```

```
  while (1)
    {
      sock_nuevo =
            accept (sock_fd, (struct sockaddr *) &cliaddr, &len);

      if (sock_nuevo < 0)
      {
        printf ("Error en la funcion accept\n");
        exit (0);
      }

      char *readbuf;
      int cantidadEnviada;
      int cantidadRecibida;

      readbuf = malloc (100);

      printf ("-----------------------------------------------\n");
      printf ("DATOS DE LA RECEPCION\n");
      printf ("-----------------------------------------------\n");

      while (1)
      {
        bzero (readbuf, 100);

        //Recibiendo segmento del cliente
        cantidadRecibida = read (sock_nuevo, readbuf, 100);

        if (cantidadRecibida == (-1))
          {
            printf ("Falla al recibir los bytes\n");
          }

        //El cliente termina la comunicacion
        if (cantidadRecibida == 0)
          {
            printf ("\n Cliente termina la comunicacion \n\n");
            close (sock_nuevo);
            free (readbuf);
            break;
          }

        printf ("\n");
        printf ("Número de bytes recibidos:%d\n",
              cantidadRecibida);
        printf ("Direccion IP remota:%s\n",
              inet_ntoa (cliaddr.sin_addr));

        //Enviando un segmento de regreso para el cliente
        cantidadEnviada =
              writeN (sock_nuevo, readbuf, cantidadRecibida);

        if (cantidadEnviada == (-1))
          {
            printf ("Falla al enviar los bytes\n");
          }
      }                 //fin while 1
    }

  return;
}                       //Fin EchoServer TCP
```

```
//------------------------------------------------------------------
//EchoServer usando SCTP
//------------------------------------------------------------------
void
echoSCTP ()
{

  int sock_fd;
  int erro;

  struct sockaddr_in servaddr, cliaddr;
  struct sctp_sndrcvinfo sri;
  struct sctp_event_subscribe evnts;

  sock_fd = socket (AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);

  if (sock_fd < 0)
    {
      printf ("Falla de creacion del socket\n");
      exit (0);
    }

  bzero (&servaddr, sizeof (servaddr));
  servaddr.sin_family = AF_INET;
  inet_aton (direccionLocal, &servaddr.sin_addr);
  servaddr.sin_port = htons (puertoLocal);

  bzero (&evnts, sizeof (evnts));
  evnts.sctp_data_io_event = 1;

  erro = setsockopt (sock_fd, IPPROTO_SCTP, SCTP_EVENTS,
                &evnts, sizeof (evnts));

  if (erro != 0)
    {
      printf ("Error al configurar la opcion de eventos en SCTP\n");
      exit (0);
    }

  struct sctp_initmsg initmsg;

  initmsg.sinit_num_ostreams = 30;
  initmsg.sinit_max_instreams = 30;
  initmsg.sinit_max_attempts = 3;
  initmsg.sinit_max_init_timeo = 30;

  int desconecta = 1;

  erro = setsockopt (sock_fd, IPPROTO_SCTP, SCTP_NODELAY,
                &desconecta, sizeof (desconecta));

  if (erro == (-1))
    {
      printf ("Error en el socket SCTP\n");
      exit (0);
    }

  erro = setsockopt (sock_fd, IPPROTO_SCTP, SCTP_INITMSG,
                (char *) &initmsg,
```

```
                sizeof (struct sctp_initmsg));

  if (erro != 0)
    {
      printf ("Error al configurar la opcion initmsg en SCTP\n");
      exit (0);
    }

  tamanoBuffer = tamanoBuffer * 1000;

  if (tamanoBuffer != 0)
    {
      erro = setsockopt (sock_fd, IPPROTO_SCTP, SO_RCVBUF,
                  (char *) &tamanoBuffer,
                  sizeof (tamanoBuffer));

      if (erro != 0)
      {
        printf ("Error al configurar en el socket la opcion de recibir
buffer\n");
        exit (0);
      }

      erro = setsockopt (sock_fd, IPPROTO_SCTP, SO_SNDBUF,
                  (char *) &tamanoBuffer,
                  sizeof (tamanoBuffer));

      if (erro != 0)
      {
        printf ("Error al configurar en el socket la opcion de envio
de buffer\n");

        exit (0);
      }
    }                   //fin tamano del buffer

  erro = bind (sock_fd, (struct sockaddr *) &servaddr,
              sizeof (servaddr));

  if (erro != 0)
    {
        printf ("Error al crear la funcion bind\n");
      exit (0);
    }

  erro = listen (sock_fd, 1);

  if (erro != 0)
    {
      printf ("Error en la funcion listen\n");
      exit (0);
    }

  socklen_t len;
  int msg_flags;
  char *readbuf;
  int cantidadEnviada;
  int cantidadRecibida;

  len = sizeof (struct sockaddr_in);
  readbuf = malloc (105);
```

```
  while (1)
    {
      bzero (readbuf, 105);

      //Recibiendo mensajes del cliente
      cantidadRecibida = sctp_recvmsg (sock_fd, readbuf, 105,
                               (struct sockaddr *)
                               &cliaddr, &len, &sri,
                               &msg_flags);

      if (cantidadRecibida == (-1))
      {
        printf ("Falla al recibir los mensajes\n");
      }

      bzero (mensagem.sendline, 100);
      memcpy (&mensagem, readbuf, cantidadRecibida);

      //DATOS DE LA COMUNICACION
      printf ("\n");
      printf ("Id:%d\n", mensagem.id);
      printf ("buffer:%s\n", mensagem.sendline);
      printf ("Numero de stream:%d\n", sri.sinfo_stream);
      printf ("Numero de bytes recibidos:%d\n",
            cantidadRecibida);
      printf ("DireccionIP remota:%s\n",
            inet_ntoa (cliaddr.sin_addr));

      //enviando un mensaje de regreso para el cliente
      cantidadEnviada =
            sctp_sendmsg (sock_fd, readbuf, cantidadRecibida,
                      (struct sockaddr *) &cliaddr,
                      sizeof (struct sockaddr),
                      sri.sinfo_ppid, mensajeDesordenado,
                      sri.sinfo_stream, 0, 0);

      if (cantidadEnviada == (-1))
      {
        printf ("Falla al enviar los mensajes\n");
      }
    }                   //fin while 1

  free (readbuf);
  return;
}                       //fin EchoServer SCTP


//------------------------------------------------------------------
//Revisando los datos de la linea de comandos
//------------------------------------------------------------------
void
verificaEntrada (void)
{
  if ((direccionLocal == NULL) ||
      (inet_aton (direccionLocal, NULL) == 0))
    {
      printf ("\nFalto colocar la Direccion IP local\n");
      uso ();
      exit (0);
    }
```

```
  if (puertoLocal == 0)
    {
      printf ("\nFalto colocar el puerto local\n");
      uso ();
      exit (0);
    }

  printf ("\n");
  printf ("Direccion IP del host local:%s\n", direccionLocal);
  printf ("Puerto del host local:%d\n", puertoLocal);
  printf ("Protocolo de transporte = ");

  if (tipoProtocolo == 1)
    printf ("TCP\n");
  else
    printf ("SCTP\n");

  if (tamanoBuffer == 0)
    printf ("Buffer \n");
  else
    printf ("Buffer de envio y recibimiento (Kbytes):%d\n",
          tamanoBuffer);

  if (mensajeDesordenado == 0)
    printf ("Entrega de mensajes: ordenada\n");
  else
    printf ("Entrega de mensajes: desordenada\n");

  printf ("\n");

}                       //fin Revisando


//---------------------------------------------------------------------
//Adquiriendo los datos de la linea de comandos
//---------------------------------------------------------------------
void
parser (int argc, char **argv)
{
  int getoptC;
  int bufferAux;
  int desorden;
  int protocolo;

  while (argc)
    {
      getoptC = getopt (argc, argv, "H:P:t:b:d:");

      if (getoptC < 0)
      break;

      switch (getoptC)
      {
      case 'H':
        direccionLocal = optarg;
        break;

      case 'P':
        puertoLocal = atoi (optarg);
        break;
```

```
      case 't':
        protocolo = atoi (optarg);
        if ((protocolo == 1) || (protocolo == 2))
          tipoProtocolo = protocolo;
        break;

      case 'b':
        bufferAux = atoi (optarg);
        if ((bufferAux > 0) && (bufferAux < 1001))
          tamanoBuffer = bufferAux;
        break;

      case 'd':
        desorden = atoi (optarg);
        if (desorden == 1)
          mensajeDesordenado = MSG_UNORDERED;
        break;

      default:
        break;
      }                 //fin switch
    }             //fin while argc
}                       //fin de parser

//---------------------------------------------------------------
//Imprime la forma de uso de la aplicacion o help
//---------------------------------------------------------------
void
uso (void)
{
  printf ("\nUso:\n");
  printf ("./EchoServer -H dirIPLoc -P ptLoc -t tipProt -b buffer -d
orden\n");
  printf (" -H dirIPLoc       - direccionIP del host local\n");
  printf (" -P ptLoc          - puerto del host local\n");
  printf (" -t tipProt        - protocolo de transporte - TCP=1 -
SCTP=2\n");
  printf (" -b buffer         - buffer de envio y recepcion(Kbytes)
min 1 max 1000 ");
  printf (" -d orden          - 1 - envia los mensajes de forma
ordenada, usar solo en SCTP\n");
  printf ("\n");
}                       //fin forma de uso

//---------------------------------------------------------------
//Funcion para forzar la escritura de todos los bytes TCP
//---------------------------------------------------------------
int
writeN (int fd, const void *buffer, int n)
{
  int nRestante;
  int nEscritos;
  const char *ptr;

  ptr = buffer;
  nRestante = n;

  while (nRestante > 0)
    {
      if ((nEscritos = write (fd, ptr, nRestante)) <= 0)
```

```
      {
        if (nEscritos < 0 && errno == EINTR)
          nEscritos = 0;
        else
          return (-1);
      }

      nRestante -= nEscritos;
      ptr += nEscritos;
    }

  return n;
}                       //fin de writeN

//-------------------------------------------------------------------
//Fin de EchoServer.c
//-------------------------------------------------------------------
```

Anexo 2

Resultados de las pruebas del Capítulo IV

Pruebas con Iperf 1.6.5

Tabla A2.1 Pruebas con Iperf 1.6.5

	TCP		SCTP	
Tiempo	Transfer (MB)	Bandwidth (Mbps)	Transfer (MB)	Bandwidth (Mbps)
10	112	94.1	110	92.5
10	112	94.1	109	91.7
10	112	94.2	110	92.4
10	112	94	110	92
10	112	94.1	110	92.5
10	112	94.2	110	92.8
10	112	94.1	109	91.8
10	112	94.2	110	92.4
10	112	94	109	91.7
10	112	93.8	110	92.5
10	112	94.2	111	93
10	112	94.1	110	92.4
10	112	93.9	109	91.7
10	112	94	110	92.8
10	112	94.1	110	92.5
10	112	94.1	110	92.4
10	112	94	110	92.5
10	112	94.2	110	92.4

Figura A2.1 SCTP vs TCP Cantidad de datos transmitidos

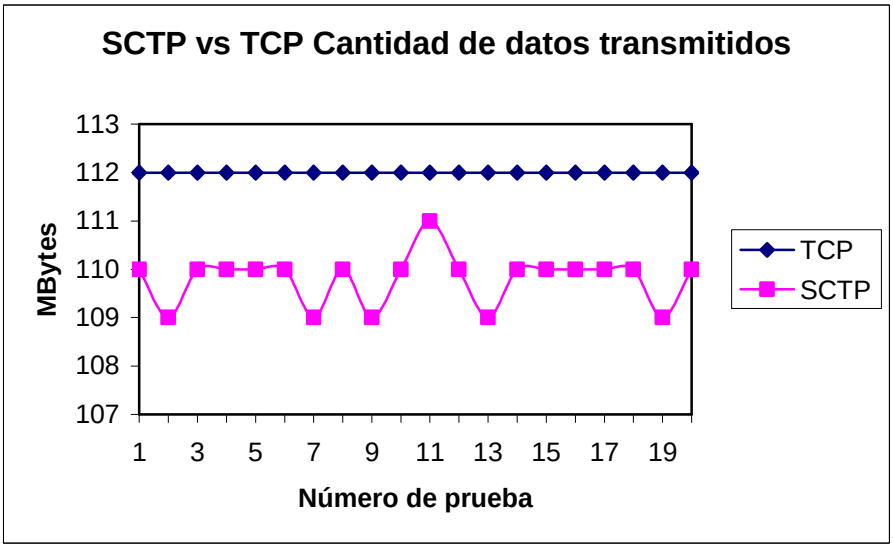


Figura A2.2 SCTP vs TCP Tasa de Transfernecia

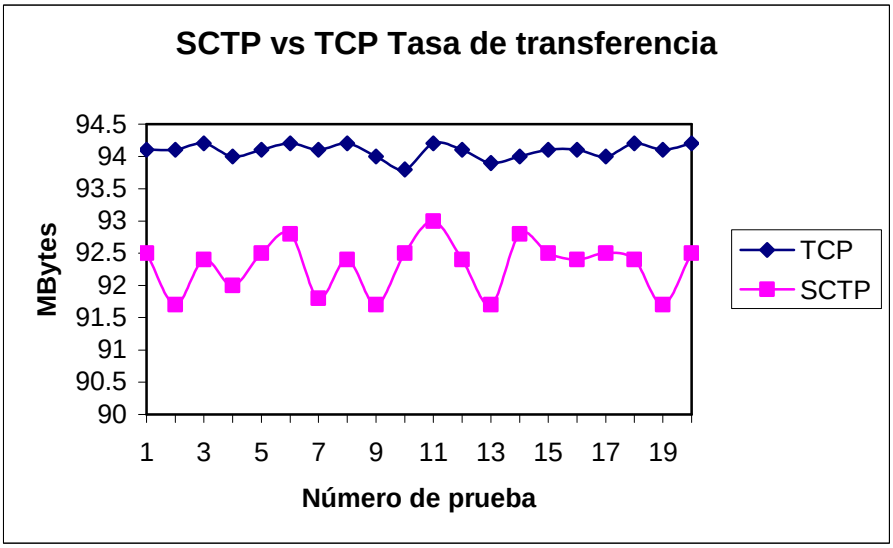


Figura A2.3 Salida del terminal usando iperf 1.6.5

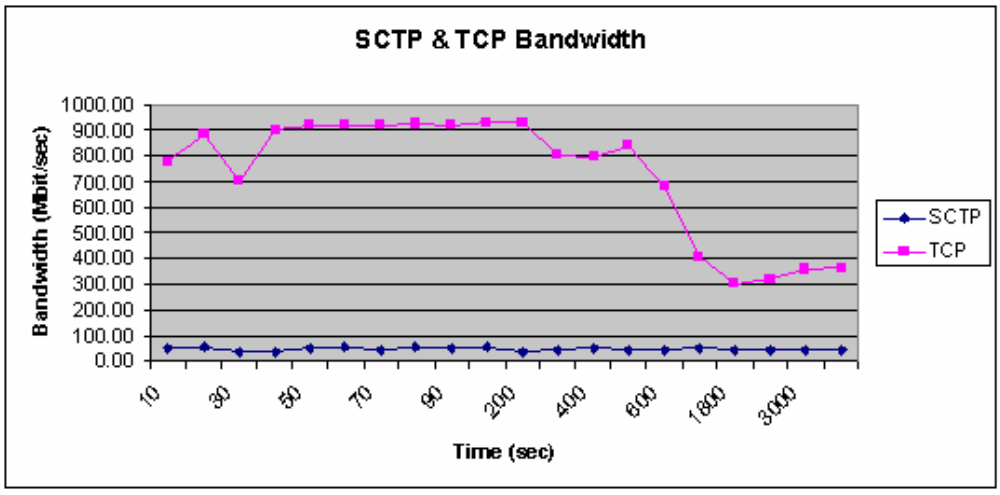
```
isaac@isaac-laptop:~/iperf-1.6.5-hb-lksctp$ ./iperf -c 192.168.203.222
In TCP mode
------------------------------------------------------------
Client connecting to 192.168.203.222, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.203.111 port 40986 connected with 192.168.203.222 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   112 MBytes  94.1 Mbits/sec
close failed: Bad file descriptor

isaac@isaac-laptop:~/iperf-1.6.5-hb-lksctp$ ./iperf -c 192.168.203.222 -z
sctp selected
In SCTP mode
------------------------------------------------------------
Client connecting to 192.168.203.222, SCTP port 5001
TCP window size:  108 KByte (default)
------------------------------------------------------------
[  3] local 192.168.203.111 port 32768 connected with 192.168.203.222 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   110 MBytes  92.5 Mbits/sec
close failed: Bad file descriptor
```

Tabla A2.2 Pruebas con Iperf 1.6.5 desde Geneva hacia Chicago de un enlace de 1Gbps

LKSCTP	SCTP		TCP	
Time	Transfer (Mbytes)	Bandwidth (Mbit/s)	Transfer (Mbytes)	Bandwidth (Mbit/s)
10	63.9	52.3	921	777
20	127	53	2060	882
30	138	38.6	2480	705
40	177	37	4190	899
50	308	51.3	5340	918
60	383	53.3	6420	917
70	351	42	7520	922
80	511	53.4	8610	925
90	552	51.5	9650	921
100	640	53.5	10800	930
200	863	36.2	21700	930
300	1490	42.6	32500	801
400	2420	52	37100	796
500	2460	42.2	49000	842
600	2990	42.8	47200	676
1200	6810	48.8	57000	408
1800	8960	42.7	63600	303
2400	12500	44.6	90500	324
3000	15700	45	125000	357
3600	18740	44.25	152000	363

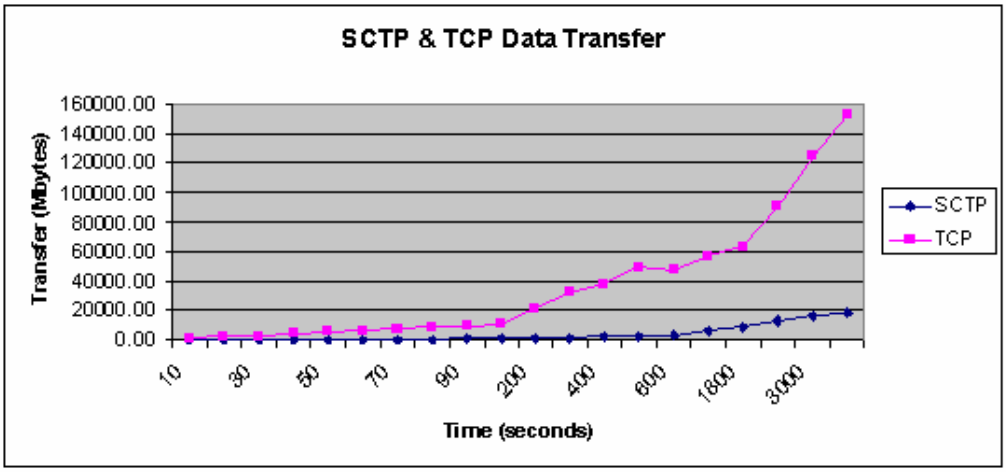
Tomado de http://datatag.web.cern.ch/datatag/WP3/sctp/tests.htm

Figura A2.3 SCTP vs TCP Cantidad de datos transmitidos (Enlace Geneva-Chicago)



Tomado de http://datatag.web.cern.ch/datatag/WP3/sctp/tests.htm

Figura A2.4 SCTP vs TCP Tasa de transferencia (Enlace Geneva-Chicago)



Tomado de http://datatag.web.cern.ch/datatag/WP3/sctp/tests.htm

Pruebas con EchoTools1.0

✓Resultados de la variación del tamaño de mensajes -Latencia Media (seg)

Tabla A2.3 Variando el tamaño de mensajes -Latencia Media (seg)

Latencia Media (seg)

	10 bytes	20 bytes	30 bytes	40 bytes	50 bytes	60 bytes	70 bytes	80 bytes	90 bytes	100 bytes
TCP	0.0059	0.0060	0.0061	0.0077	0.0093	0.0079	0.0081	0.0082	0.0071	0.0061
SCTP 1 stream	0.0025	0.0026	0.0027	0.0032	0.0038	0.0043	0.0048	0.0053	0.0045	0.0036
SCTP 10 streams	0.0043	0.0024	0.0004	0.0004	0.0004	0.0025	0.0035	0.0046	0.0050	0.0054
SCTP 50 streams	0.0005	0.0004	0.0004	0.0004	0.0004	0.0030	0.0043	0.0056	0.0050	0.0043
SCTP 100 streams	0.0004	0.0003	0.0003	0.0018	0.0034	0.0033	0.0041	0.0048	0.0040	0.0032
SCTP 150 streams	0.0005	0.0005	0.0004	0.0004	0.0003	0.0004	0.0004	0.0004	0.0004	0.0004
SCTP 200 streams	0.0010	0.0007	0.0003	0.0010	0.0016	0.0007	0.0006	0.0004	0.0004	0.0004
SCTP 250 streams	0.0004	0.0004	0.0004	0.0013	0.0022	0.0008	0.0006	0.0004	0.0005	0.0006
SCTP 300 streams	0.0003	0.0008	0.0013	0.0015	0.0017	0.0038	0.0049	0.0061	0.0059	0.0057

Figura A2.4 Variando el tamaño de mensajes -Latencia Media

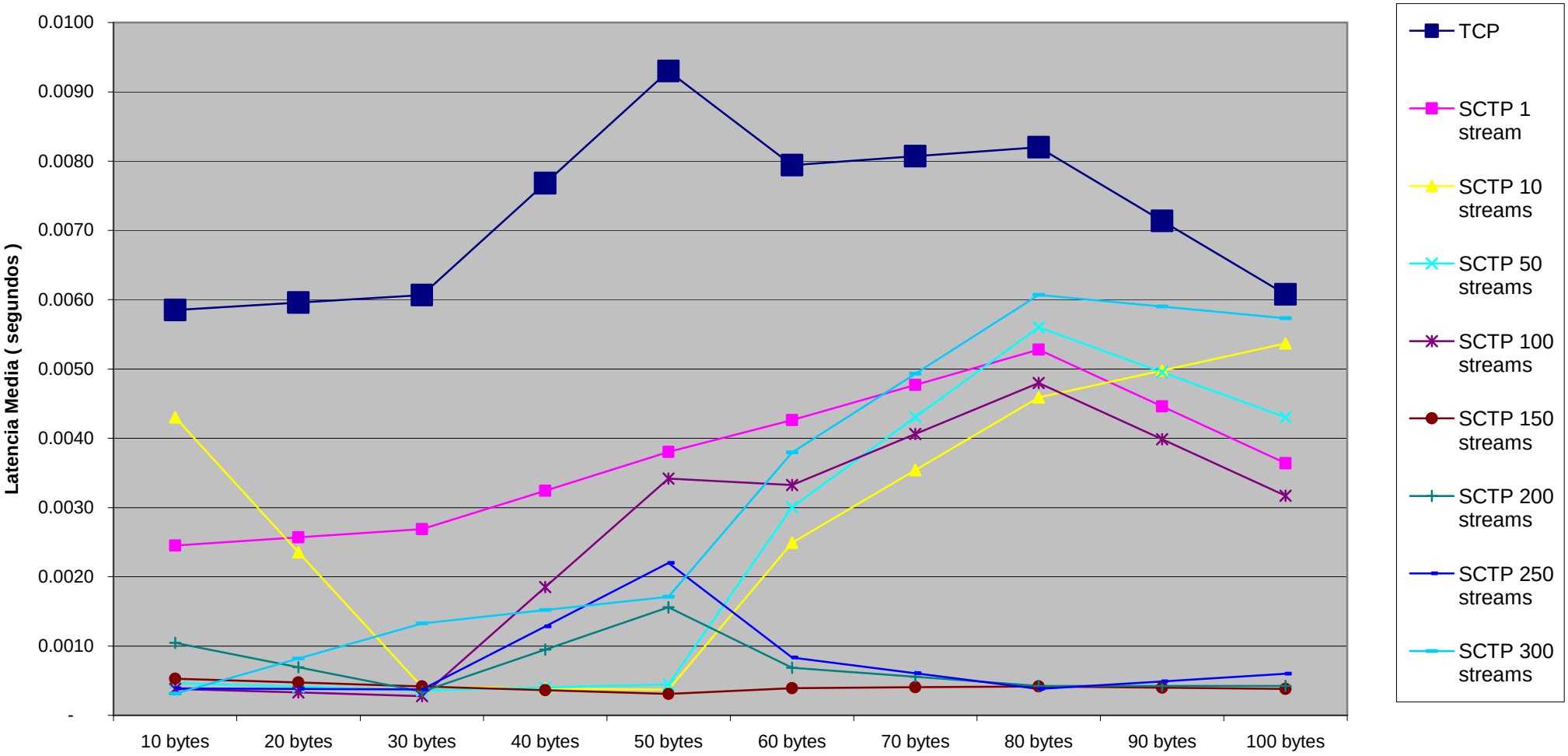


Figura A2.5 Variando el tamaño de mensajes -Latencia Media

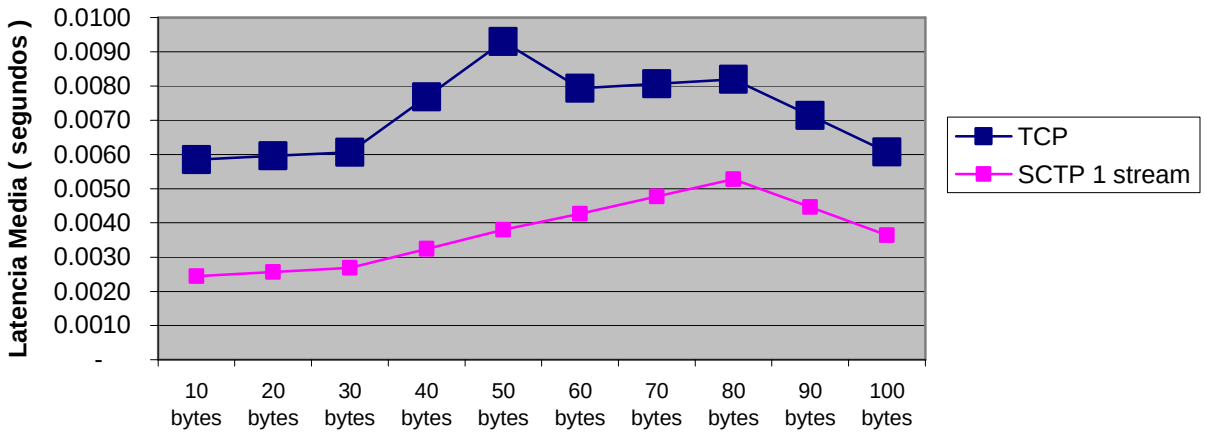


Figura A2.6 Variando el tamaño de mensajes -Latencia Media

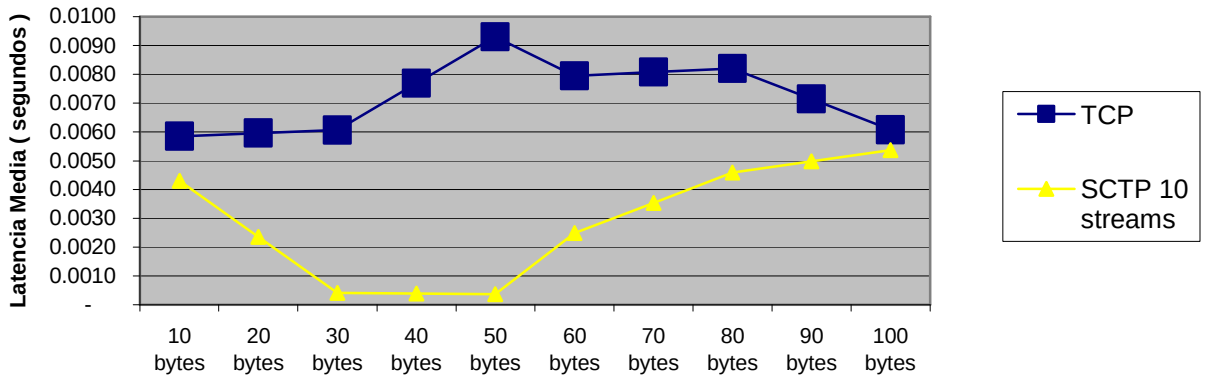


Figura A2.7 Variando el tamaño de mensajes -Latencia Media

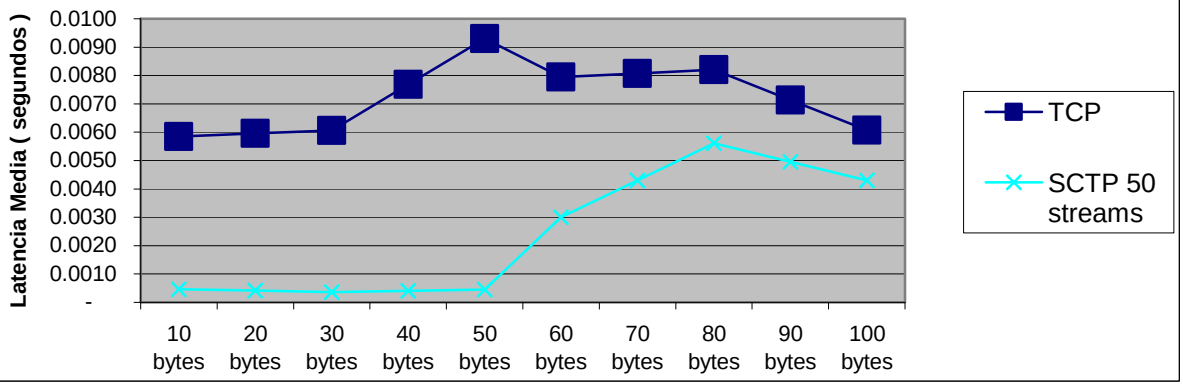


Figura A2.8 Variando el tamaño de mensajes -Latencia Media

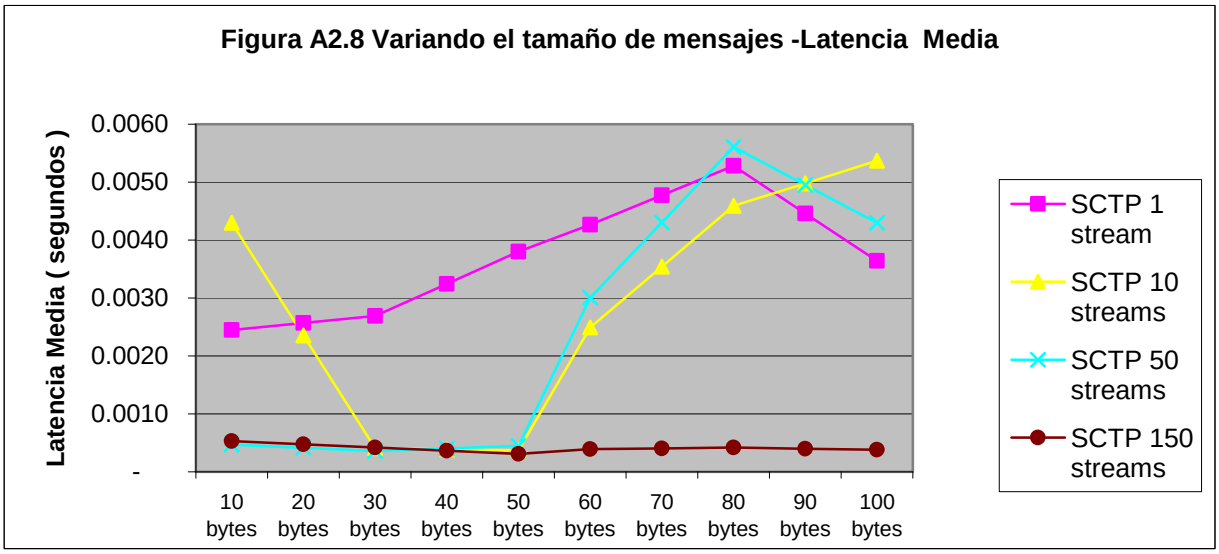
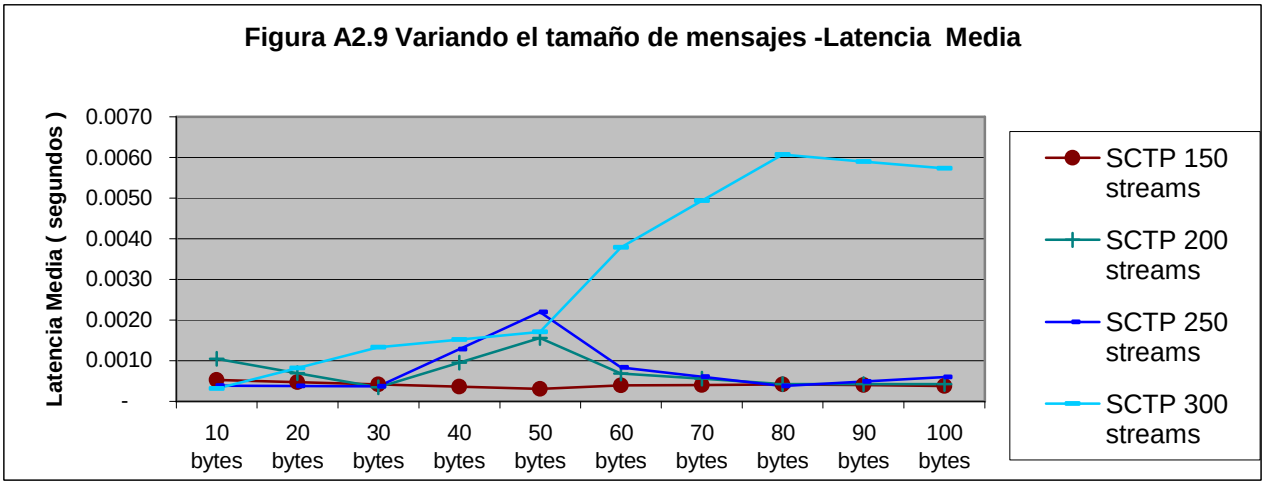


Figura A2.9 Variando el tamaño de mensajes -Latencia Media



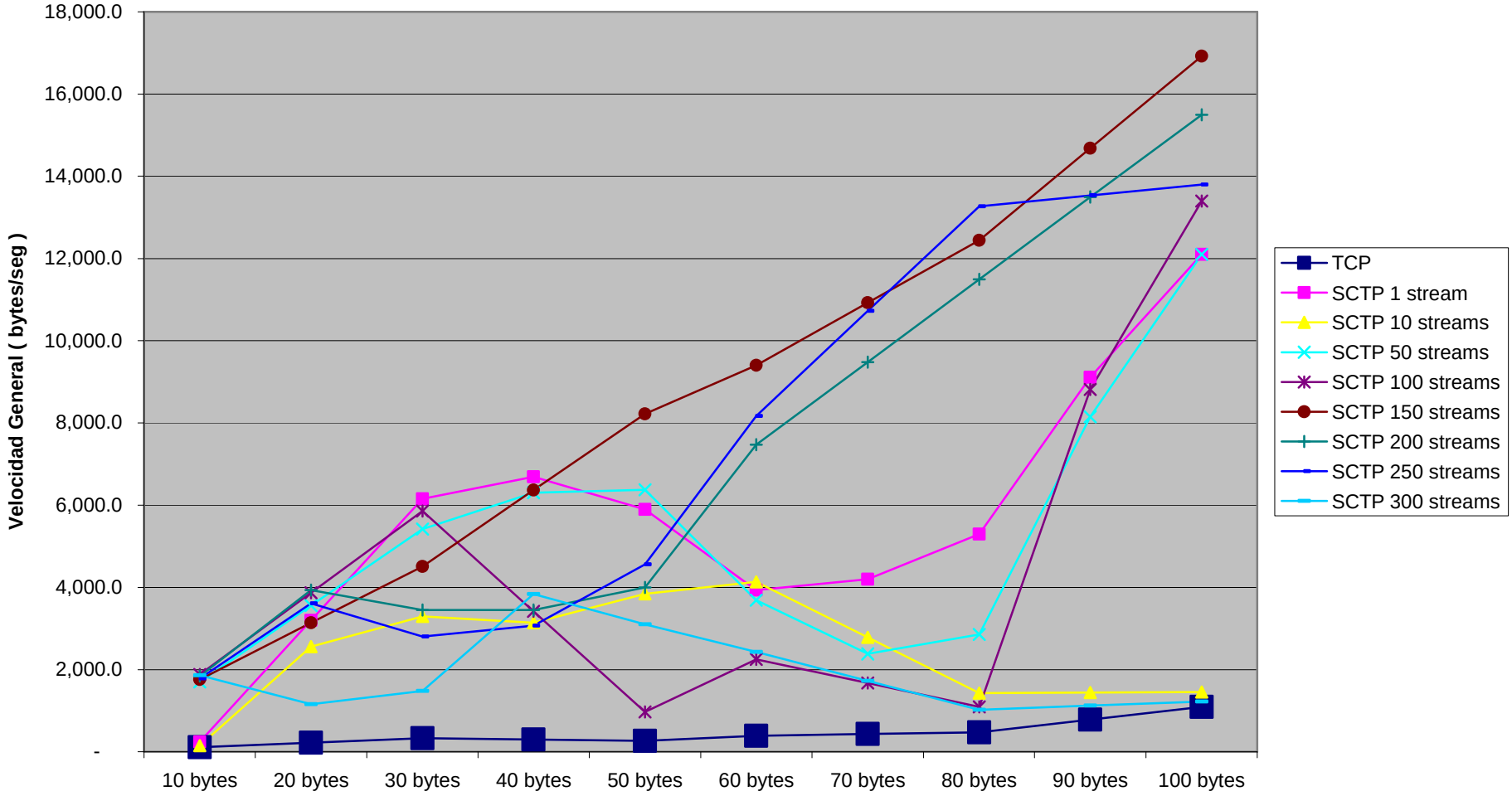
✓ Resultados de la variación del tamaño de mensajes - Velocidad General (kbytes/seg)

Tabla A2.4 Variando el tamaño de mensajes –Velocidad General (kbytes/seg)

Velocidad General (Kbps)

	10 bytes	20 bytes	30 bytes	40 bytes	50 bytes	60 bytes	70 bytes	80 bytes	90 bytes	100 bytes
TCP	108.5	218.0	327.4	297.3	267.1	386.2	430.7	475.2	785.2	1,095.2
SCTP 1 stream	238.5	3,196.2	6,153.9	6,688.4	5,900.0	3,934.9	4,200.0	5,300.0	9,114.2	12,100.0
SCTP 10 streams	148.6	2,551.8	3,290.0	3,140.0	3,840.0	4,130.4	2,780.5	1,430.5	1,442.9	1,455.4
SCTP 50 streams	1,699.6	3,560.5	5,421.5	6,300.9	7,180.3	3,688.2	2,381.8	1,075.5	8,147.8	12,100.0
SCTP 100 streams	1,884.0	3,870.9	5,857.8	3,413.0	968.2	2,253.6	1,673.9	1,094.1	8,815.1	13,400.0
SCTP 150 streams	1,759.9	3,135.5	4,511.2	6,364.8	8,218.4	9,403.9	10,923.4	12,443.0	14,683.9	16,924.8
SCTP 200 streams	1,841.3	3,934.1	6,026.8	3,444.9	863.0	7,468.6	9,480.5	11,492.3	13,495.5	15,498.7
SCTP 250 streams	1,785.3	3,612.0	5,438.7	3,066.2	693.7	8,170.1	10,722.0	13,273.9	13,535.1	13,796.2
SCTP 300 streams	1,863.2	1,158.7	454.3	3,832.7	3,100.0	2,427.8	1,725.4	1,023.0	1,123.7	1,224.5

Figura A2.10 Variando el tamaño de mensajes - Velocidad General



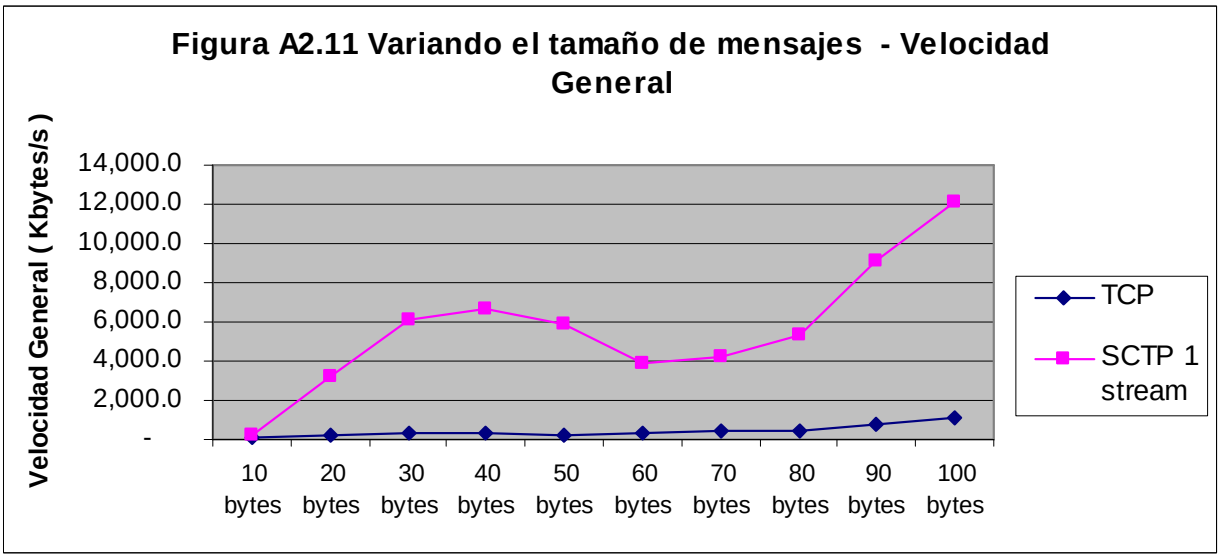


Figura A2.11 Variando el tamaño de mensajes - Velocidad General
Velocidad General (Kbytes/s)
14,000.0
12,000.0
10,000.0
8,000.0
6,000.0
4,000.0
2,000.0
-
10 bytes
20 bytes
30 bytes
40 bytes
50 bytes
60 bytes
70 bytes
80 bytes
90 bytes
100 bytes
TCP
SCTP 1 stream

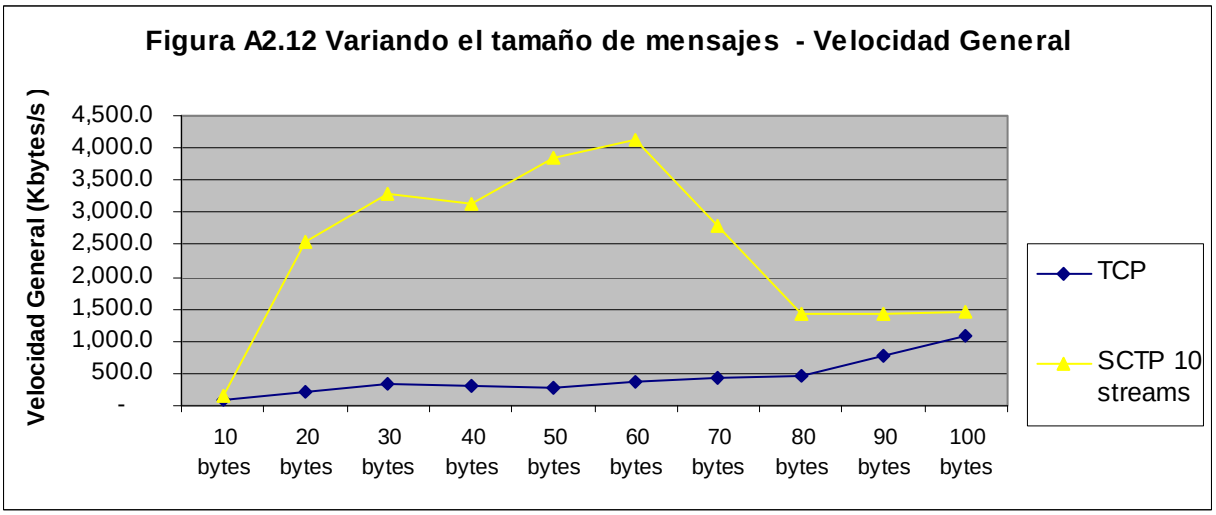


Figura A2.12 Variando el tamaño de mensajes - Velocidad General
Velocidad General (Kbytes/s)
4,500.0
4,000.0
3,500.0
3,000.0
2,500.0
2,000.0
1,500.0
1,000.0
500.0
-
10 bytes
20 bytes
30 bytes
40 bytes
50 bytes
60 bytes
70 bytes
80 bytes
90 bytes
100 bytes
TCP
SCTP 10 streams

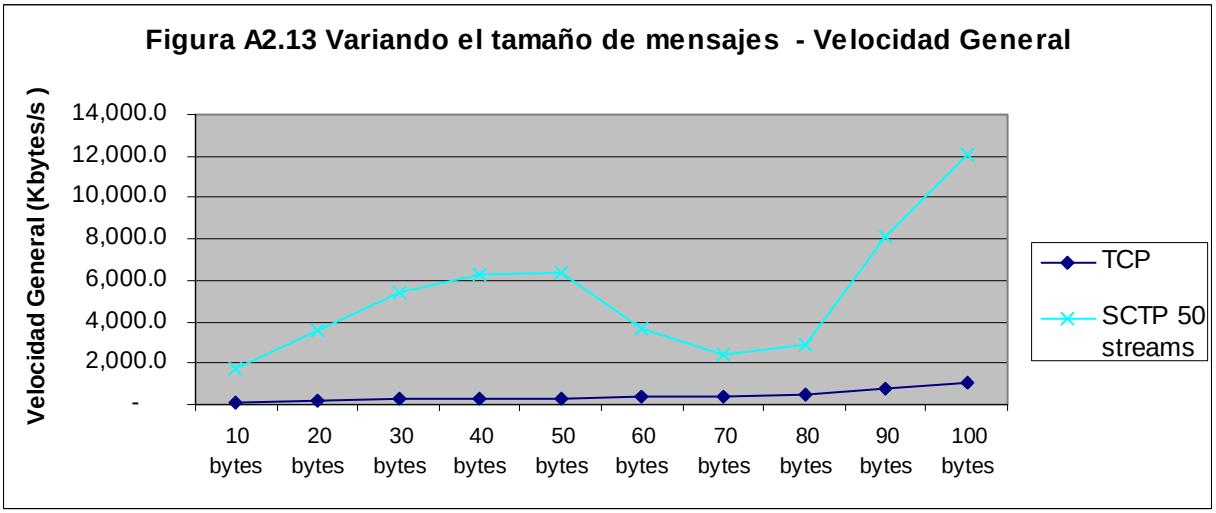


Figura A2.13 Variando el tamaño de mensajes - Velocidad General
Velocidad General (Kbytes/s)
14,000.0
12,000.0
10,000.0
8,000.0
6,000.0
4,000.0
2,000.0
-
10 bytes
20 bytes
30 bytes
40 bytes
50 bytes
60 bytes
70 bytes
80 bytes
90 bytes
100 bytes
TCP
SCTP 50 streams

Figura A2.14 Variando el tamaño de mensajes - Velocidad General

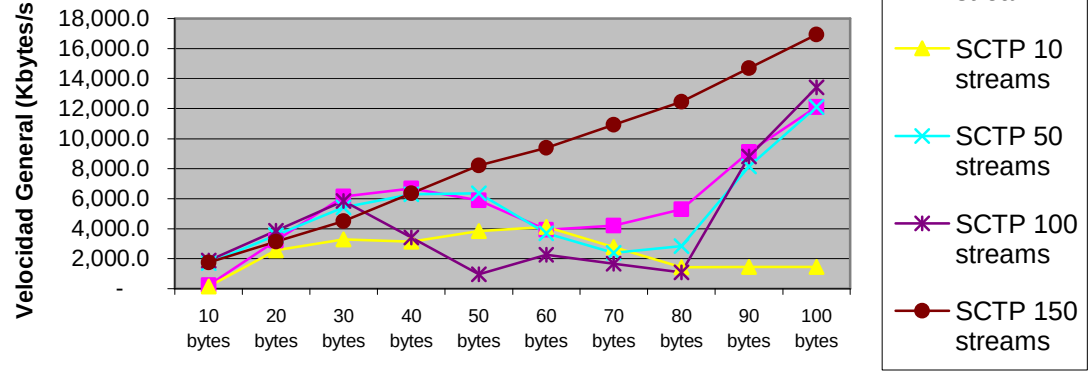
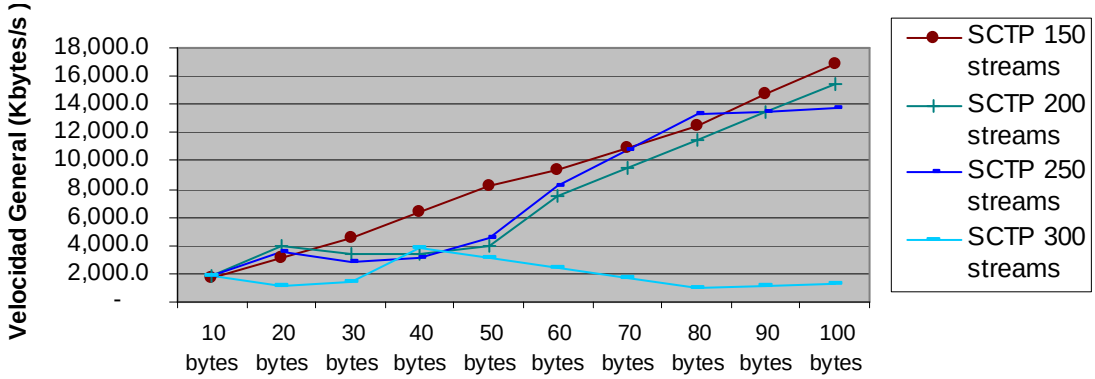


Figura A2.15 Variando el tamaño de mensajes - Velocidad General



✓Resultados de la variación del número de mensajes – Latencia Media(seg)

Tabla A2.5 Variando el número de mensajes -Latencia Media (seg)

Latencia Media (seg)

	10 msg	20 msg	30 msg	40 msg	50 msg	60 msg	70 msg	80 msg	90 msg	100 msg
TCP	0.0101	0.0120	0.0138	0.0128	0.0117	0.0112	0.0104	0.0097	0.0084	0.0071
SCTP 1 stream	0.0042	0.0046	0.0050	0.0051	0.0053	0.0050	0.0050	0.0049	0.0048	0.0047
SCTP 10 streams	0.0008	0.0023	0.0038	0.0021	0.0004	0.0012	0.0008	0.0004	0.0010	0.0016
SCTP 50 streams	0.0026	0.0033	0.0040	0.0022	0.0004	0.0013	0.0009	0.0004	0.0004	0.0004
SCTP 100 streams	0.0096	0.0087	0.0077	0.0061	0.0045	0.0058	0.0056	0.0055	0.0029	0.0004
SCTP 150 streams	0.0007	0.0006	0.0005	0.0032	0.0058	0.0042	0.0048	0.0053	0.0044	0.0035
SCTP 200 streams	0.0007	0.0006	0.0005	0.0005	0.0005	0.0028	0.0039	0.0051	0.0047	0.0043
SCTP 250 streams	0.0007	0.0006	0.0005	0.0005	0.0004	0.0027	0.0039	0.0050	0.0046	0.0042
SCTP 300 streams	0.0008	0.0038	0.0069	0.0057	0.0045	0.0051	0.0048	0.0045	0.0048	0.0052

Figura A2.16 Variando el número de los mensajes - Latencia Media

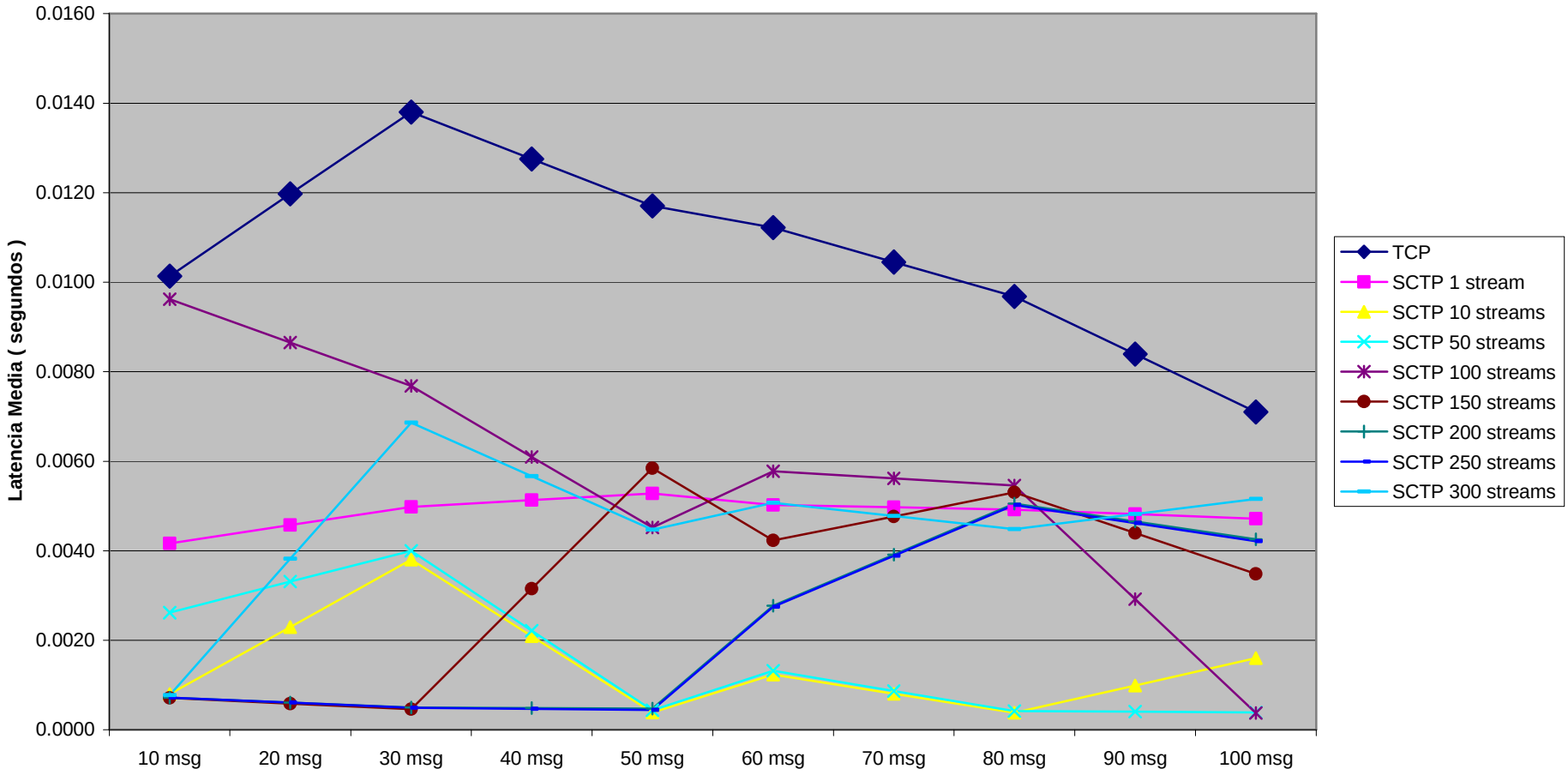


Figura A2.17 Variando el número de los mensajes - Latencia Media

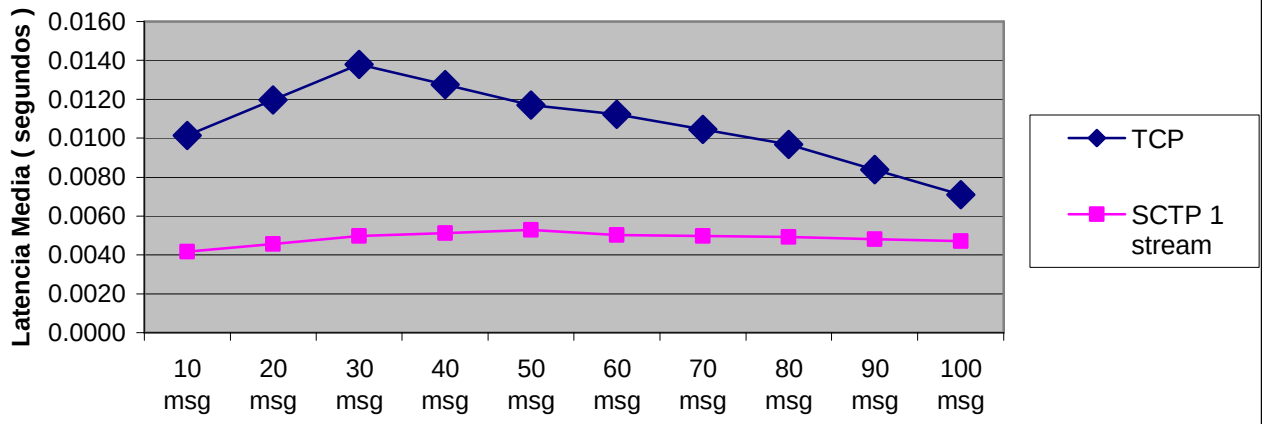


Figura A2.18 Variando el número de los mensajes - Latencia Media

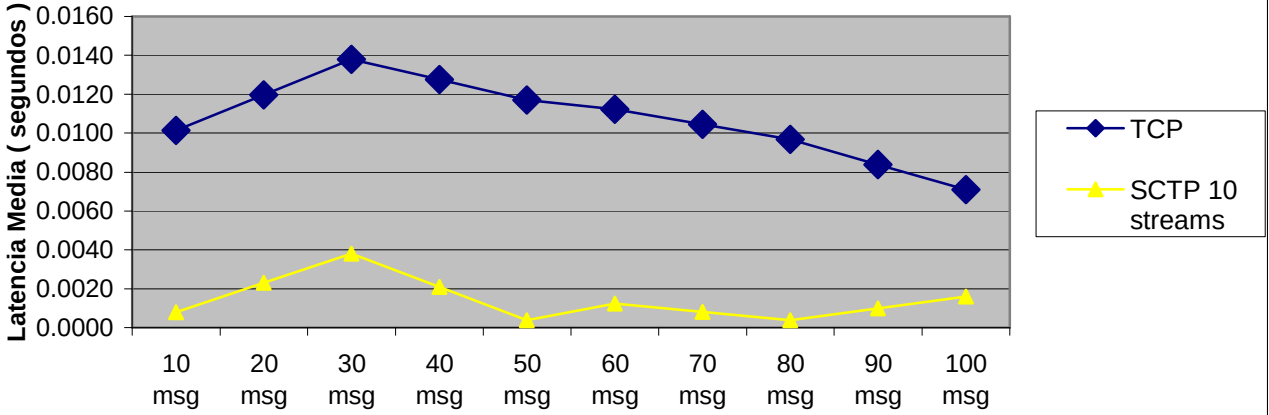
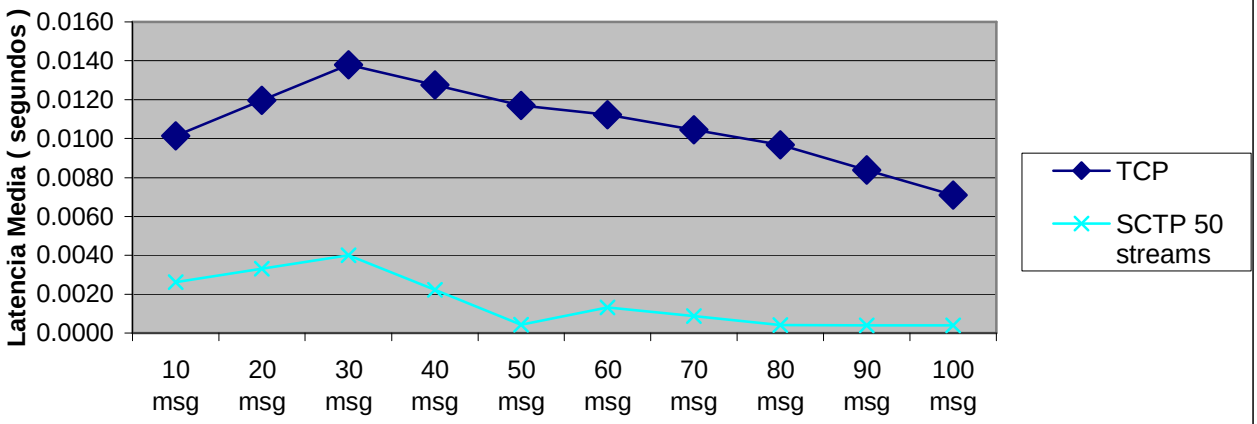


Figura A2.19 Variando el número de los mensajes - Latencia Media



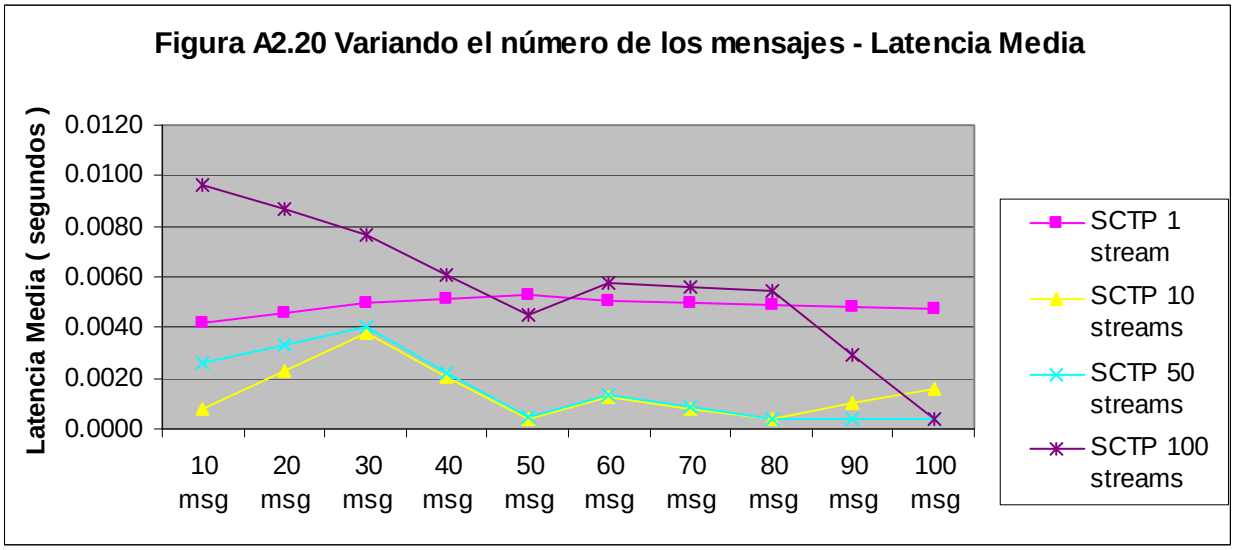


Figura A2.20 Variando el número de los mensajes - Latencia Media
Latencia Media (segundos)
0.0120
0.0100
0.0080
0.0060
0.0040
0.0020
0.0000
10 msg
20 msg
30 msg
40 msg
50 msg
60 msg
70 msg
80 msg
90 msg
100 msg
SCTP 1 stream
SCTP 10 streams
SCTP 50 streams
SCTP 100 streams

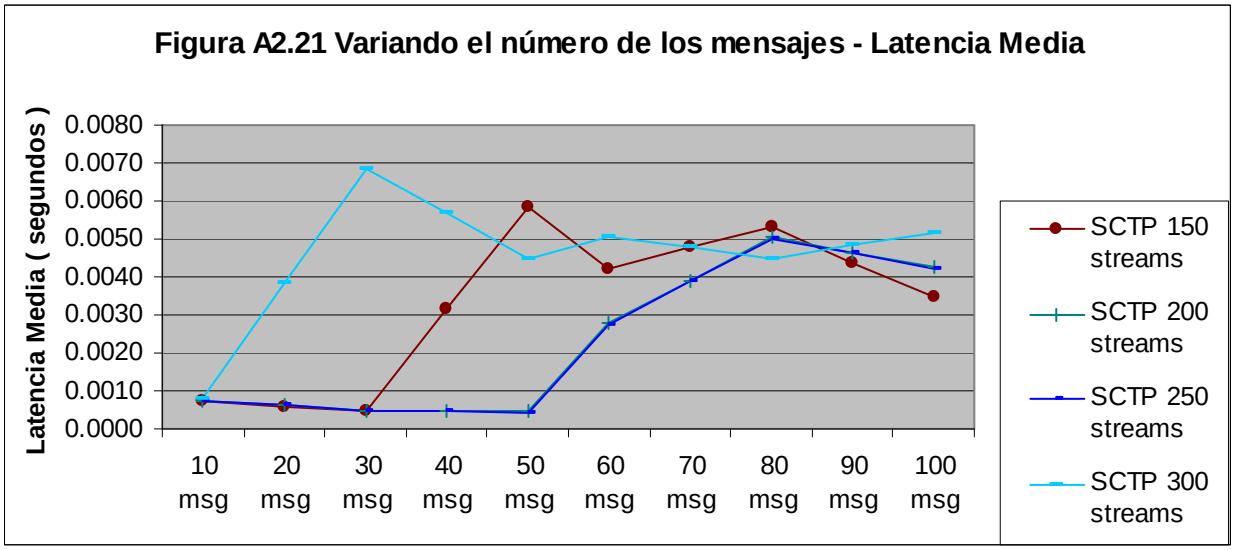


Figura A2.21 Variando el número de los mensajes - Latencia Media
Latencia Media (segundos)
0.0080
0.0070
0.0060
0.0050
0.0040
0.0030
0.0020
0.0010
0.0000
10 msg
20 msg
30 msg
40 msg
50 msg
60 msg
70 msg
80 msg
90 msg
100 msg
SCTP 150 streams
SCTP 200 streams
SCTP 250 streams
SCTP 300 streams

✓ Resultados de la variación del número de mensajes - Velocidad General (kbytes/seg)

Tabla A2.6 Variando el número de mensajes - Velocidad General (kbytes/s)

Velocidad General (Kbps)

	10 msg	20 msg	30 msg	40 msg	50 msg	60 msg	70 msg	80 msg	90 msg	100 msg
TCP	143	179	216	671	1127	783	839	894	1543	2192
SCTP 1 stream	1255	1480	1705	1557	1409	1836	1975	2115	2393	2671
SCTP 10 streams	2211	1441	671	8570	16470	16795	20907	25020	24000	27700
SCTP 50 streams	995	926	857	7841	14826	15402	19182	22962	26166	29370
SCTP 100 streams	512	618	724	1347	1971	1623	1762	1900	16327	30755
SCTP 150 streams	2964	6420	9875	5470	1064	3754	2896	2038	16781	31524
SCTP 200 streams	2718	5371	8025	10702	13379	6371	4206	2040	2991	3941
SCTP 250 streams	2688	5416	8144	10741	13338	6584	4506	2427	3313	4198
SCTP 300 streams	2288	1610	932	1614	2295	1910	2059	2207	2414	2622

Figura A2.22 Variando el número de los mensajes - Velocidad General

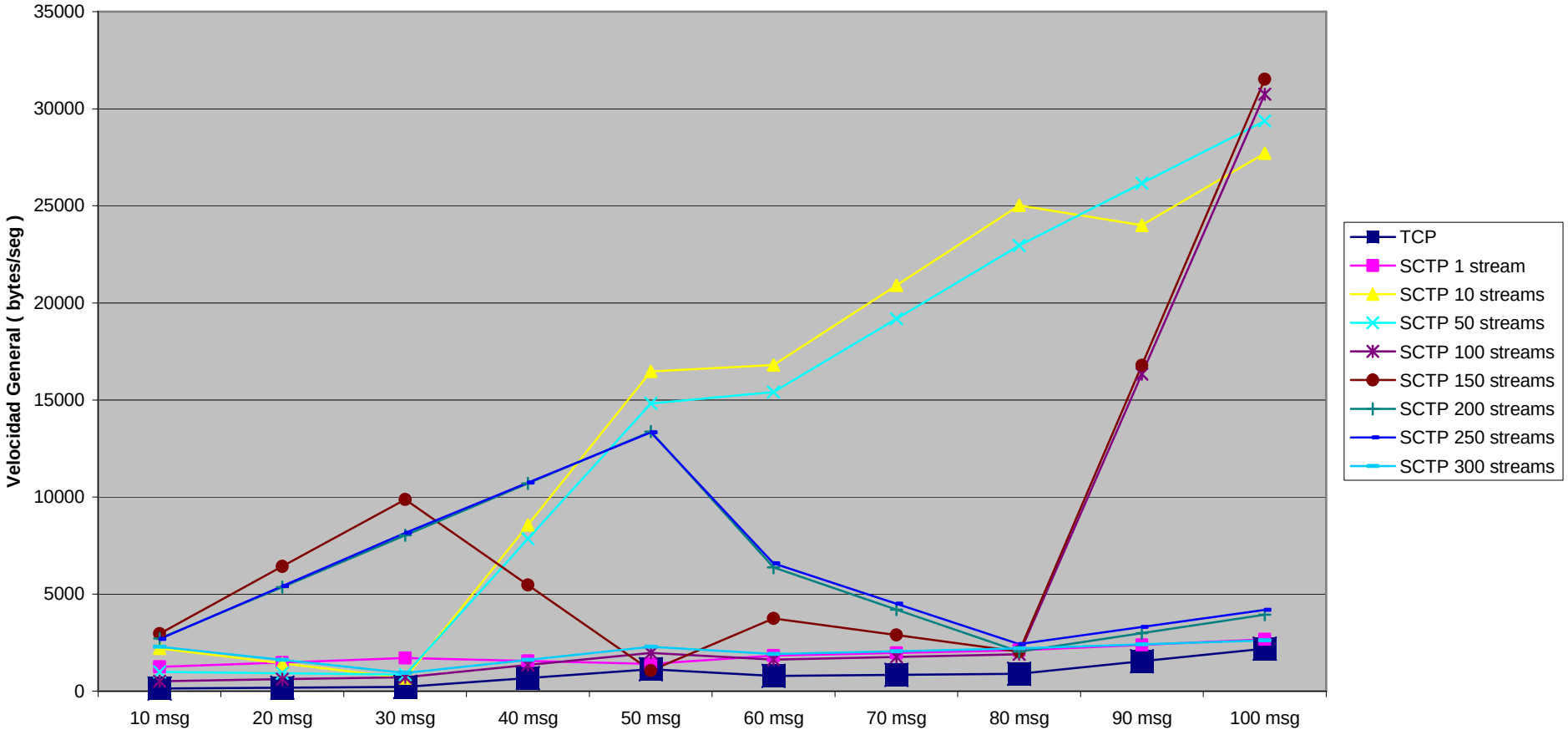


Figura A2.23 Variando el número de los mensajes - Velocidad General

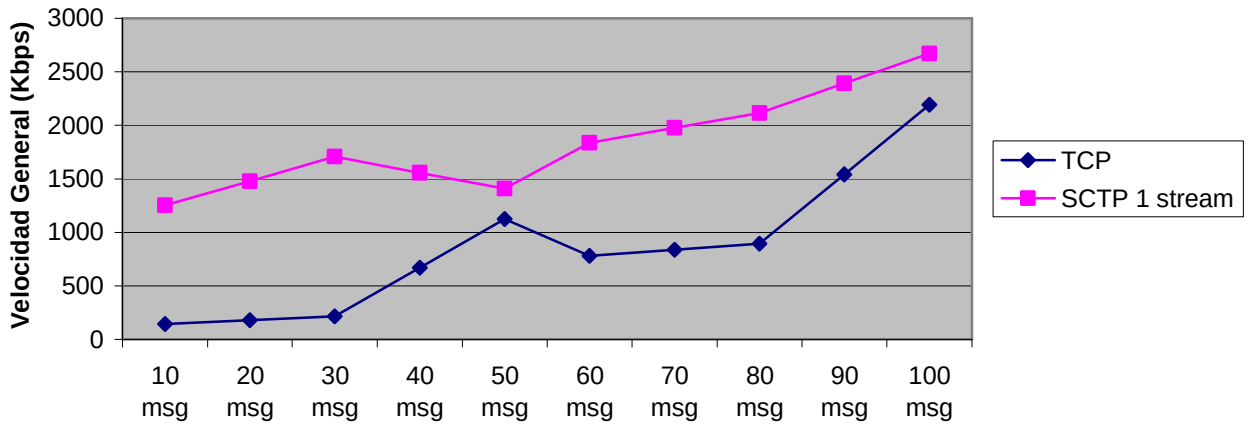


Figura A2.24 Variando el número de los mensajes - Velocidad General

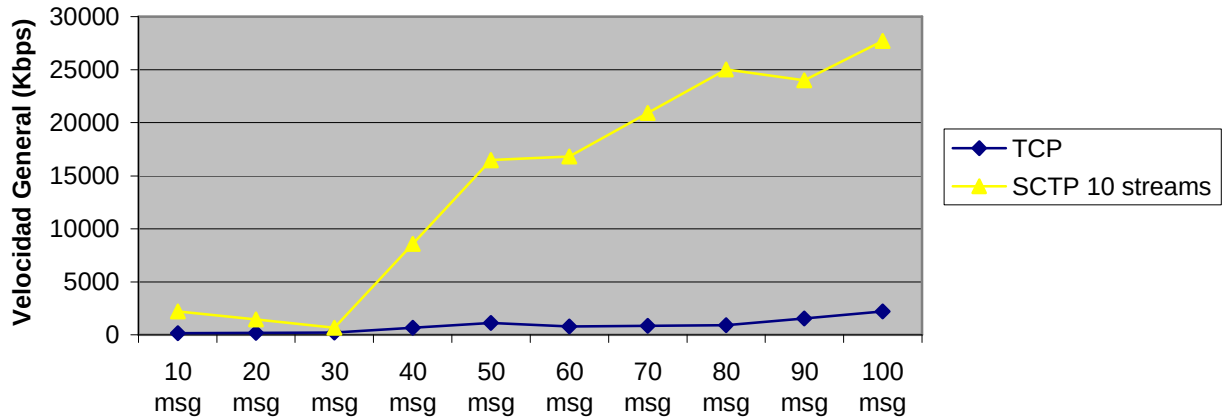


Figura A2.25 Variando el número de los mensajes - Velocidad General

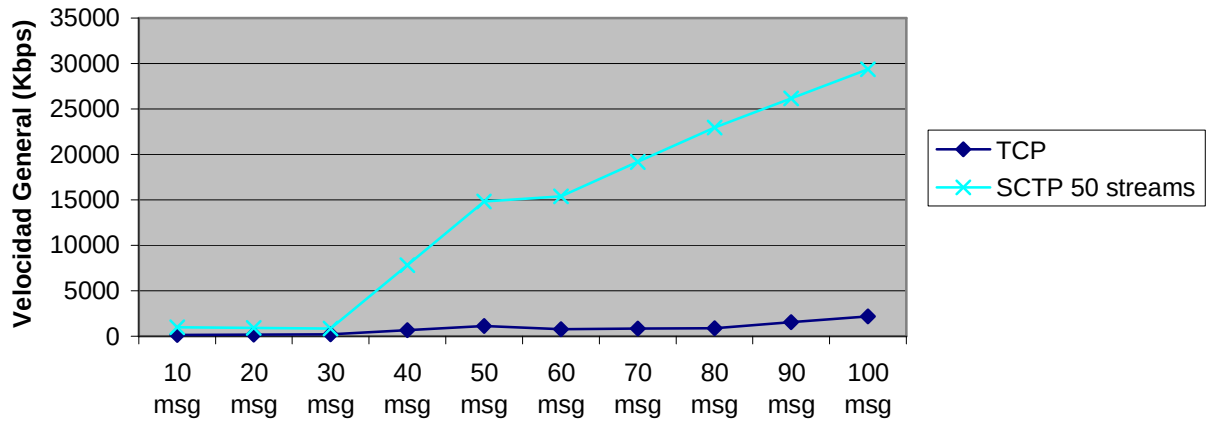


Figura A2.26 Variando el número de los mensajes - Velocidad General

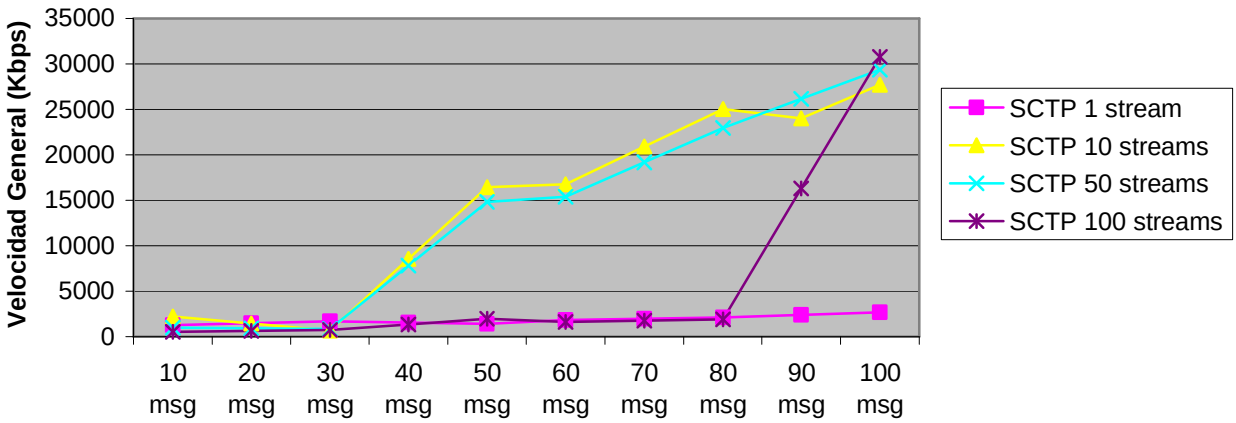


Figura A2.27 Variando el número de los mensajes - Velocidad General

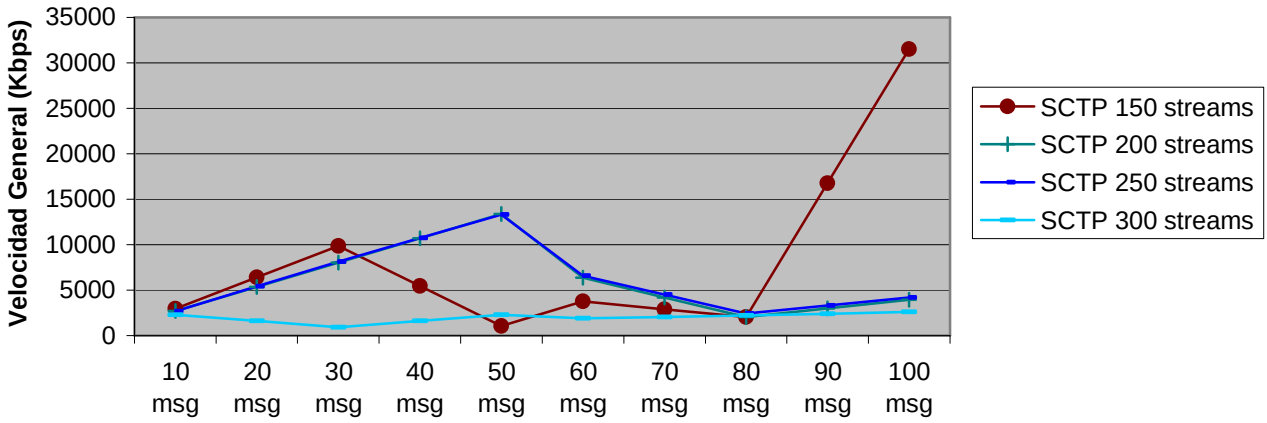


Figura A2.28 ESCENARIO DE PRUEBAS EN EL LABORATORIO V203-PUCP

Figura A2.29 ESCENARIO DE PRUEBAS EN EL LABORATORIO V203-PUCP

En el caso del cliente utilizamos un PC con las siguientes características:

Laptop Dell Inspiron 710m
Procesador Pentium M de 2GHz
Memoria RAM de 1GB
Tarjeta de red Broadcom 440x 10/100 Integrated

En el caso del servidor usamos un PC con las siguientes características:

Laptop Acer
Procesador Celeron 1.7GHz
Memoria RAM de 512MB
Tarjeta de red Broadcom 440x 10/100 Integrated

En el caso del conmutador utilizamos uno con las siguientes características:

Accelar 1200
48 puertos 10/100 Integrated

Pruebas con SCTPperf1.0

Figura A2.30 SCTPperf sobre ethernet

```
isaac@isaac-laptop:~/sctpperf_0.1$ ./scptperf_clnt -P 1111 -H 192.168.203.111 -p 2222 -h
192.168.203.222 -l 1000 -t 5 -x 1 -m 50 -f 100 >>clnt11.txt
Local host:               192.168.203.111
Local port:               1111
Remote host:              192.168.203.222
Remote port:              2222
Packet size:              1000
Measure period:           5
Print period:             1
Streams:                  50
Sender buffer:            102400
Receiver buffer:          102400
Verbose:                  0
Binding single address
[2007/07/19 18:35:33] 1s_BDW = 88.054214 Mbits/s, PACKETS = 11543, TIME = 1 sec 135 usec
[2007/07/19 18:35:34] 1s_BDW = 87.807480 Mbits/s, PACKETS = 11510, TIME = 1 sec 78 usec
[2007/07/19 18:35:35] 1s_BDW = 87.666130 Mbits/s, PACKETS = 11491, TIME = 1 sec 37 usec
[2007/07/19 18:35:36] 1s_BDW = 87.804855 Mbits/s, PACKETS = 11509, TIME = 1 sec 21 usec
[2007/07/19 18:35:37] 1s_BDW = 87.805382 Mbits/s, PACKETS = 11509, TIME = 1 sec 15 usec
BANDWIDTH = 87.827621 Mbits/s, SIZE = 1000 bytes, SENT = 57562 packets
```

Figura A2.31 SCTPperf sobre wireless (802.11b)

```
isaac@isaac-laptop:~/sctpperf_0.1$ ./scptperf_clnt -P 1111 -H 192.168.0.100 -p 2222 -h
192.168.0.101 -l 1000 -t 5 -x 1 -m 50 -f 100 >>clnt12.txt
Local host:               192.168.0.100
Local port:               1111
Remote host:              192.168.0.101
Remote port:              2222
Packet size:              1000
Measure period:           5
Print period:             1
Streams:                  50
Sender buffer:            102400
Receiver buffer:          102400
Verbose:                  0
Binding single address
[2007/07/19 18:42:10] 1s_BDW = 0.380367 Mbits/s, PACKETS = 150, TIME = 3 sec 8699 usec
[2007/07/19 18:42:11] 1s_BDW = 0.966947 Mbits/s, PACKETS = 127, TIME = 1 sec 2054 usec
[2007/07/19 18:42:12] 1s_BDW = 1.042338 Mbits/s, PACKETS = 137, TIME = 1 sec 2772 usec
BANDWIDTH = 0.630010 Mbits/s, SIZE = 1000 bytes, SENT = 414 packets
```

Figura A2.32 SCTPperf sobre ethernet y *wireless* (802.11b) - *Multihoming*

```
flavio@flavio-laptop:~/Isaac_Tesis/sctpperf_0.1$ ./scptperf_srv -P 2222 -H 192.168.203.222 -B
192.168.0.101 -f 100

Local host:               192.168.203.222
Local port:               2222
Sender buffer:            102400
Receiver buffer:          102400
Verbose:                  0
Binding single address
Binding mutliple addresses from addr_buf[]
[2007/07/19 21:30:38.203262] 1s_BDW = 83.436073 Mbits/s, PACKETS = 422, TIME = 1 sec 5840 usec
[2007/07/19 21:30:39.209948] 1s_BDW = 85.639572 Mbits/s, PACKETS = 360, TIME = 1 sec 6686 usec
[2007/07/19 21:30:40.215061] 1s_BDW = 85.773598 Mbits/s, PACKETS = 360, TIME = 1 sec 5113 usec
[2007/07/19 21:30:41.219981] 1s_BDW = 85.790070 Mbits/s, PACKETS = 360, TIME = 1 sec 4920 usec
[2007/07/19 21:31:45.930857] 1s_BDW = 0.353699 Mbits/s, PACKETS = 60, TIME = 64 sec 710876 usec
BANDWIDTH = 5.300504 Mbits/s, SIZE = 100000 bytes, RECEIVED = 1568 packets
```

Anexo 3

INDICE

1. Algunos enrutadores inalámbricos basados en Software Libre.....2

2. Chipsets usados en tarjetas inalámbricas. Algunas marcas y funcionamiento general. 2

3. Ventajas de las soluciones inalámbricas basadas en Software Libre 3

4. Comandos de configuración de tarjetas inalámbricas con Software Libre 3

5. Métodos de configuración de enrutamiento en enrutadores con Software Libre.....7

REFERENCIAS 8

1. Algunos enrutadores inalámbricos basados en Software Libre

Los enrutadores basados en software libre ofrecen funciones para operar como *bridge*, punto de acceso y router simultáneamente, utilizan bajo consumo de energía entre 4 y 5watts, generalmente vienen en una tarjeta madre, en la cual se pueden conectar tarjetas inalámbricas como pueden ser las de *chipset Prism* que son las más utilizadas por los usuarios de Linux, así mismo utilizan una memoria la cual puede ser una *SD, Standard Disk o Compact Flash*, la cual tiene una ranura especial para ser insertada, en esta memoria se instala el sistema operativo. Utilizan también un procesador, el cual puede ser un AMD.

Los enrutadores más conocidos son los de placa *Soekris* NET4521 y NET4511, están basados en un procesador x86 AMD, y utilizan una *Compact Flash*, y utilizan las tarjetas inalámbricas *Atheros* o *Prism*, las cuales pueden ser PCI o *miniPCI*.

2. Chipsets usados en tarjetas inalámbricas. Algunas marcas y funcionamiento general.

El "***chipset***" es el conjunto (*set*) de chips que se encargan de controlar determinadas funciones de un dispositivo, en este caso la tarjeta inalámbrica

En el mercado tenemos varios tipos de *chipset*, entre ellos: Chipset Prism, Chipset Hermes, Chipset Symbol, Chipset Atheros, Chipser Broadcom.

Chipset Prism

Este es uno de los más utilizados por la comunidad GNU/Linux, y ya que toda la información referente a estos se puede conseguir fácilmente, eso ha conllevado que se creen diversos controladores para este chipset como son las herramientas linux-wlan-ng, controladores Prism54, Airjack , controladores HostAP.

En cuanto a su funcionamiento tiene un controlador de MAC que realiza la mayor parte de las operaciones básicas del protocolo 802.11, posee un motor WEP que agiliza el trabajo de criptografía.

Chipset Aironet

A Partir del *chipset Prism Cisco* añadió algunas características como permitir el paso de la banda ISM a otra frecuencia y control de potencia, sin embargo con el tiempo estos dos *chipset* sean diferentes, este *chipset* tiene una excelente sensibilidad a la recepción y permite monitorización.

Chipset Hermes

Este *chipset* fue desarrollado por *Lucent*, es de código cerrado, aun que *Lucent* libero parte del código fuente para las funciones básicas del controlado Orinoco. Este *chipset* identifica decide el *essid* ni bien se activa la interfaz inalámbrica, posee el modo monitor aun que es necesario añadir un parche.

3. Ventajas de las soluciones inalámbricas basadas en Software Libre

La principal ventaja es que se reducen los costos ya que no se necesita pagar por licencia, gracias a la licencia GNU/Linux, y por lo mismo que se puede acceder al código fuente lo que permite, que nuevos usuarios puedan estudiar el software, copiarlo y modificarlo, redistribuirlos, y mejorarlo.
De esta manera se consigue que los demás usuarios cooperen para que se mejore el software, y se puedan dividir las tareas para arreglar los *bugs*. Así se crean nuevas mejoras como es el caso del *chipset Prism*, el cual tiene varios controladores.
Tienen más servicios como es el caso de Linux AP que ofrece *telnet, httpd* y ha mejorado las opciones iniciales del punto de acceso comercial como son modo master, cliente, repetidor, *bridge*, WDS, *aodv, openvpn*.

4. Comandos de configuración de tarjetas inalámbricas con Software Libre

Para configurar una tarjeta inalámbrica es necesario asegurarnos que la tarjeta esta siendo reconocida por la interfaz, para eso utilizamos el comando *iwconfig*.

```
isaac@isaac-laptop:~$ iwconfig
lo no wireless extensions.

eth0 no wireless extensions.

eth1 IEEE 802.11b/g ESSID:"wifipucp" Nickname:"Broadcom 4318"
Mode:Managed Frequency=2.484 GHz Access Point: Invalid
Bit Rate=1 Mb/s Tx-Power=18 dBm
RTS thr:off Fragment thr:off
Link Quality:0 Signal level:0 Noise level:0
Rx invalid nwid:0 Rx invalid crypt:0 Rx invalid frag:0
Tx excessive retries:0 Invalid misc:0 Missed beacon:0

sit0 no wireless extensions.

isaac@isaac-laptop:
```

Como vemos en este *host* la interfaz eth1 ha sido reconocida como interfaz inalámbrica, en caso no fuera reconocida aparecería como *no wireless extensions* como es el caso de las demás interfaces.

Sin embargo, la tarea de hacer reconocer una tarjeta inalámbrica no es tan sencilla, y algunos casos puede resultar un dolor de cabeza para muchos.

El problema es que muchos fabricantes tienen convenios con Windows y producen sus controladores solo para este sistema operativo, por lo que existe algunos grupos de desarrolladores que programan controladores compatibles para Linux.

Uno de las aplicaciones más conocidas es el *ndiswrapper,* el cual es una herramienta que nos ayuda a reconocer varios tipos de tarjetas inalámbricas, haciendo uso del controlador que es utilizado en Windows.

Para el caso a ejemplificar es una tarjeta inalámbrica *Dell Wireless MiniPCI 1370* con *chipset Broadcom Corporation BCM4318 [AirForce One 54g] 802.11g Wireless LAN Controller (rev 02),* para poder ver el tipo de *chipset* de nuestra tarjeta utilizamos el comando

```
$lspci
```

Descargar e instar el *ndiswrapper*, si estamos usando Ubuntu podemos descargarlo de los repositorios utilizando el Synaptic o bien con el comando.

```
$sudo apt-get install ndiswrapper-utils
```

Si no se consiguiese instalar, revisar los repositorios, de ser correcta la instalación pasamos a descargar los controladores de Windows, se recomienda descargar el controlador y no copiar y pegar el mismo que esta siendo utilizado por Windows.

Debemos tener los archivos bcmwl5.inf, y bcmwl5.sys.

Bien luego de ello utilizamos el siguiente comando

```
$sudo ndiswrapper -i bcmwl5.inf
```

(Ubicarse en la carpeta donde se encuentra el *driver*)

De ser correcto este procedimiento utilizamos el siguiente comando

```
$sudo ndiswrapper -l
```

y nos debería salir:

```
Installed ndis drivers:
bcmwl5 driver present, hardware present
```

Ahora levantamos el modulo del *ndiswrapper*, con el comando.

```
$sudo modprobe ndiswrapper
$sudo ndiswrapper -m
```

Verificamos que se haya levantado correctamente con el comando

```
$lsmod | grep ndiswrapper
```

Si todo ha sucedido con éxito el LED indicador de la tarjeta inalámbrica debería haberse encendido. Si queremos utilizar el *ndiswrapper* nos faltaría hacer la configuración de la tarjeta que se detalla más adelante.

Aunque es muy útil el *ndiswrapper* ya que solamente nos proporciona el uso cotidiano de la tarjeta inalámbrica, sin embargo si queremos realizar tareas de auditoria en redes inalámbricas, se recomiendo no usar *ndiswrapper,* ya que no soporta aplicaciones como *Aircrack*. Por lo que es más conveniente levantar la tarjeta con el *driver* creado especialmente para las tarjetas inalámbricas del tipo *bcm43xx* o en su defecto de otra tarjeta que se este utilizando.

Para ello mejor desinstalamos el *ndiswrapper* desde el synaptic o bajamos el modulo.

Ahora debemos de descargar el paquete wl_apsta.o el cual contiene los *drivers* bcm4318 entre otros, desde **http://drinus.net/airport/wl_apsta.o** y el paquete *bcm43xx-fwcutter* eso lo podemos hacer con el comando:

```
$sudo apt-get install bcm43xx-fwcutter
```

Ahora corremos el paquete *bcm43xx-fwcutter* con el archivo descargado, buscar donde esta el archivo, y aplicar el siguiente comando

```
$sudo bcm43xx-fwcutter -w <downloaded file>
```

```
$sudo bcm43xx-fwcutter -w  wl_apsta.o
```

A continuación levantamos el modulo así como hicimos con el *ndiswrapper*

```
$sudo modprobe bcm43xx
```

Ahora verificamos nuevamente con el *iwconfig* para ver con que alias esta la interfaces, (ver la respuesta a este comando antes mencionada).

Ahora pasamos a configurar la tarjeta.
Primero debemos de ver las redes disponibles, para nuestro caso la interfaz inalámbrica se encuentra bajo el alias de eth1

```
sudo iwlist eth1 scan
```

Debe salir una respuesta como la siguiente:

```
isaac@isaac-laptop:-$ sudo iwlist eth1 scan
eth1      Scan completed :
          Cell 01 - Address: 00:90:4B:67:A9:F6
                    ESSID:"wifipucp"
                    Protocol:IEEE 802.11bg
                    Mode:Master
                    Channel:4
                    Encryption key:on
                    Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 6 Mb/s; 9 Mb/s
                              11 Mb/s; 12 Mb/s; 18 Mb/s; 24 Mb/s; 36 Mb/s
```

```
                    48 Mb/s; 54 Mb/s
                Quality=100/100  Signal level=-54 dBm
                Extra: Last beacon: 268ms ago
        Cell 02 - Address: 00:90:4B:1C:7B:E4
                ESSID:"wifipucp"
                Protocol:IEEE 802.11bg
                Mode:Master
                Channel:9
                Encryption key:on
                Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 6 Mb/s; 9 Mb/s
                        11 Mb/s; 12 Mb/s; 18 Mb/s; 24 Mb/s; 36 Mb/s
                        48 Mb/s; 54 Mb/s
                Quality=100/100  Signal level=-67 dBm
                Extra: Last beacon: 148ms ago
isaac@isaac-laptop:-$
```

Ahora por ejemplo si deseamos conectarnos a la red *wifipucp*, configuramos el tipo de red, el essid, una clave WEP.

Agregando el essid:

```
isaac@isaac-laptop:~$ sudo iwconfig eth1 essid 'wifipucp'
```

Ponemos en modo Infraestructura:

```
isaac@isaac-laptop:~$ sudo iwconfig eth1 mode Managed
```

Agregamos la clave WEP: **********

```
isaac@isaac-laptop:~$ sudo iwconfig eth1 key restricted **********
```

Luego de hacer cambios en la interfaz para que hagan efecto debemos reiniciarla.

```
isaac@isaac-laptop:~$ sudo ifconfig eth1 down
isaac@isaac-laptop:~$ sudo ifconfig eth1 up
```

Y como esta red utiliza DHCP, utilizamos el siguiente comando para obtener la dirección IP, mascara, DNS.
isaac@isaac-laptop:~$ sudo dhclient eth1

```
isaac@isaac-laptop:-$ sudo dhclient eth1
Internet Systems Consortium DHCP Client V3.0.4
Copyright 2004-2006 Internet Systems Consortium.
All rights reserved.
For info, please visit http://www.isc.org/sw/dhcp/

Listening on LPF/eth1/00:14:a5:50:4b:81
Sending on LPF/eth1/00:14:a5:50:4b:81
Sending on Socket/fallback
DHCPDISCOVER on eth1 to 255.255.255.255 port 67 interval 4
DHCPOFFER from 192.168.30.10
DHCPREQUEST on eth1 to 255.255.255.255 port 67
DHCPACK from 192.168.30.10
bound to 192.168.30.211 -- renewal in 1417 seconds.
isaac@isaac-laptop:-$
```

Y listo nos hemos conectado a una red, también podemos editar el archivo */etc/network/interfaces* para no configurar a cada momento.

```
isaac@isaac-laptop:~$ sudo vi /etc/network/interfaces
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet dhcp
auto eth1
iface eth1 inet dhcp
wireless-essid wifipucp
wireless-key **********
auto eth2
iface eth2 inet dhcp
```

Finalmente estamos listos para utilizar los servicios de la WLAN.

5. Métodos de configuración de enrutamiento en enrutadores con Software Libre.

Enrutamiento Estático, viene por incluido en el kernel de Linux, se muestra como el paquete IPROUTE, aun que es posible obtenerlo de la siguiente dirección ftp://ftp.inr.ac.ru/ip-routing, o en su defecto de los repositorios. Este paquete nos proporciona comandos para realizar la administración del trafico IP, configurar las interfaces, protocolos, se pueden realizar también monitoreo de periféricos, y el uso de tablas de enrutamiento.

Como ejemplo mostramos la asignación de direcciones ip estáticas con el comando
$*ip address add 192.168.1.115 dev eth1*

Como también podemos establecer las rutas por ejemplo modificando las tablas de enrutamiento, acá mostramos un ejemplo utilizado en el curso de Análisis de Protocolos, donde modificamos el archivo */etc/iproute2/rt_tables* agregando tablas, para luego asignarles reglas de enrutamiento.

```
# reserved values
#
#255    local
#254    main
#253    default
#0      unspec
#
# local
#
#1      inr.ruhep
100     ISP1
200     ISP2
```

A las cuales se les aplica comandos de este tipo, para la tabla ISP1 por ejemplo.

ip route add 192.168.1.0/24 dev eth1 src 192.168.1.155 table isp1
ip route add default via 192.168.1.2 table isp

Enrutamiento Dinámico, de manera similar a un router en Linux podemos implementar métodos de enrutamiento dinámico como RIP, OSPF.

Ejemplo de configuración de RIP:

Ejecutamos el demonio con /usr/sbin/router luego creamos el siguiente script y lo ejecutamos.

```
!/sbin/bash

# Habilitamos la redirección de paquetes
echo 1 > /proc/sys/net/ipv4/ip_forward

# Configuramos las interfaces
/sbin/ifconfig eth0 192.168.1.2 netmask 255.255.255.0
/sbin/ifconfig eth1 192.168.2.1 netmask 255.255.255.0
/sbin/ifconfig eth2 192.168.3.1 netmask 255.255.255.0

# Gateway por defecto
/sbin/route add default gw 192.168.1.1
# Arrancamos routed
/usr/sbin/routed
```

REFERENCIAS

[1]. Using Broadcom Wireless in Ubuntu Edgy 6.10: https://help.ubuntu.com/community/WifiDocs/Driver/bcm43xx/Edgy
[2]. Ndiswrapper: http://ndiswrapper.sourceforge.net/mediawiki/index.php/Installation
[3]. El chipset:http://www.conozcasuhardware.com/quees/chipset.htm
[4]. Aironet : http://www.cisco.com/en/US/products/hw/wireless/ps430/prod_technical_reference_list.html
[5]. Muñoz J. ,Material del Curso Análisis de Protocolos, PUCP
[6]. Chávez A. ,Material del Curso Ingeniería Inalámbrica, PUCP